(4)

RADC-TR-88-296
Final Technical Report
March 1989

FAULT TOLERANT SOFTWARE TECHNOLOGY FOR DISTRIBUTED COMPUTER SYSTEMS

Georgia Institute of Technology

Richard J. LeBlanc

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

RADC TR-88-296 has been reviewed and is approved for publication.

APPROVED:

Richard A Metzger

RICHARD A. METZGER
Project Engineer

APPROVED:

[illegible]

RAYMOND P. URTZ, JR., Technical Director
Directorate of Communications

FOR THE COMMANDER:

[illegible]

IGOR G. PLONISCH
Directorate of Plans and Programs

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-296
6a. NAME OF PERFORMING ORGANIZATION Georgia Institute of Technology; 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center(COTD)
6c. ADDRESS (City, State, and ZIP Code) Atlanta Fulton County GA 30332-0420	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center; 8b. OFFICE SYMBOL (If applicable) COTD	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-86-C-0032
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. 62702F; PROJECT NO. 5581; TASK NO 21; WORK UNIT ACCESSION NO. 77

11. TITLE (Include Security Classification)
FAULT TOLERANT SOFTWARE FOR DISTRIBUTED COMPUTER SYSTEMS

12. PERSONAL AUTHOR(S)
Richard J. LeBlanc

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM Feb 86 TO Feb 88	March 1989	80

16. SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Distributed Operating Systems, Fault Tolerant Computer Systems, Programming Methodologies and Techniques, System Architectures and Design
12	05		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report summarizes the work performed over a two-year period by the CLOUDS project at Georgia Institute of Technology to address the methodologies for fault tolerant software design and implementation in an object-oriented distributed operating system. The major research results are contained in two companion guide book reports resulting from this effort entitled, "Action Based Programming for Embedded Systems" and "Programming Techniques for Resilience and Availability." The information in this report provides an overview of the major aspects of the system and identifies the major issues which are considered in detail in the guidebooks.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL RICHARD A. METZGER	22b. TELEPHONE (Include Area Code) (315) 330-2066; 22c. OFFICE SYMBOL RADC(COTD)

DD Form 1473, JUN 86 Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

1. Summary of Project

This report documents the results of the project entitled "Fault Tolerant Software Technology for Distributed Computing Systems," a two year effort performed at Georgia Institute of Technology as part of the Clouds Project. The Clouds Project is building an object-oriented distributed operating system and studying how such a system supports the development of distributed applications, with a particular concern for highly available, fault-tolerant applications. The Clouds kernel supports *objects* as the fundamental encapsulation of data. Objects define permanent virtual address spaces and may allow access to and modification of their data through arbitrary, programmer-defined operations. Objects operations are invoked using capabilities which allow system-wide access to an object via the kernel-based capability interpretation mechanism. The kernel also provides *atomic actions* (corresponding roughly to the database notion of atomic transactions) in order to support the construction of reliable applications.

The design philosophy of the Clouds system is that the fundamental tools needed for the development of distributed applications are (1) a mechanism for distributed data access and (2) support for dealing with component failures. The object mechanism described above is designed expressly to support location-transparent data sharing. Processes interact not by passing messages to one another, but rather by accessing a shared object. This approach allows the processes to directly share the common part of their collective state rather than to attempt to communicate state changes directly to one another via messages.

Since the state of a computation is captured by the set of objects it access and modifies, it is important that component failures do not lead to inconsistent states in which some but not all of a list of related changes to objects have been completed. The atomic action mechanism provides such an assurance. An atomic action can be defined to consist of any number of operation invocations on a set of objects.

The project consisted of two research tasks. The goal of each task was the production of a technical guidebook outlining and analyzing tools and techniques for the development of fault tolerant software for distributed computing systems.

The title of Task 1 was "Programming Techniques for Resilience and Availability." The work of this task focused heavily on problems related to replication, since replication is the key ingredient of any scheme to provide highly available applications or services. Issues discussed in the guidebook include defining resilient data areas, naming replicas, locking in the presence of replicas, state propagation to replicas when actions commit, and fault tolerant action execution. Much of the discussion is in terms of the Aeolus language which we use to program objects for the Clouds system.

The title of Task 2 was "Action-Based Programming for Embedded Systems." The major issue addressed by this task is the seeming incompatibility of the idea of large-grained atomic actions with the irreversible operations frequently performed by embedded systems. Substantial consideration is given to the problem of preserving information about irreversible operations so that recovery mechanisms invoked by action aborts (or exceptions) can produce a meaningful system state, though not the same state as would be produced by a pure atomic action mechanism.

1.1. Applicability to Existing Systems

The resilience work in Task 1 focuses on features in the Aeolus language designed to support the definition of resilient objects. Its major point of general applicability is in how it relates to the more general concept of checkpointing. It illustrates the value and power of allowing a programmer to specify what must be checkpointed and how it is to be recovered.

The availability work in Task 1 is again somewhat specialized for the Clouds environment. However, it should be viewed as a model for the importance of allowing a mixture of contribution by the system and the programmer. The basic idea of the solution presented is that the system provides a basic framework and supporting mechanisms for availability while the programmer contributes policy implementations that are customized for a particular application.

The embedded system work of Task 2 has a more direct general application. It generalizes the atomicity concept by integrating forward and backward recovery, thus removing the incompatibility between the (generalized) atomicity concept and irreversible operations.

A common thread through all of the results is the importance of providing ways for a programmer to use application semantics in developing customized recovery, resilience and availability solutions, while at the same time providing the most powerful supporting mechanisms possible.

2. Programming Techniques for Resilience and Availability

In keeping with the title of this task, the most significant results presented in the Task 1 guidebook concern language features for resilient types and availability specifications. In keeping with our concern for providing powerful supporting mechanisms, it is significant to note that both of these features are *declarative*. Our intent is to allow a programmer, as far as possible, to specify resilience and availability requirements, leaving detail work to a compiler and runtime library.

Resilience and availability are crucial to our basic goal: fault tolerant software for distributed computing systems. By *resilience* we mean the survivability and consistency of data despite crashes and other detectable faults. We define *availability* to mean accessibility of data despite network partitions or failures of some sites in a distributed system. Together with a mechanism that ensures *forward progress* (continued execution of jobs despite failures), these properties provide fault tolerance.

Resilient types are a mechanism for specifying customized update and recovery mechanisms in an object designed to be modified by atomic actions. Such modifications imply that multiple versions of the object must be maintained while uncompleted actions exist. Use of customized operations based on object semantics in this context allows far more efficient use of atomic actions than would be possible if a generalized recovery scheme were used. In the later case, a copy of the entire data space of the object would have to be made for each active version of the object. Use of the resilient type technique allows all versions to be represented within a single address space.

The features for availability support presented in the guidebook are collectively known as *distributed locking*. They deal with support for managing replicas of an object in order to increase the availability of the service provided by the object. Using distributed locking, a programmer first writes a definition and implementation of an object as if only a single instance of the object was going to exist. (Resilient types might be used in the object implementation.) The programmer then writes an availability specification for the object, specifying the number of replicas, the the replication control policies to be used, and the relative availabilities of the modes of each lock type specified in the object. The most significant aspect of this specification is the replication control policy part. It allows the programmer to designate how concurrency control and consistency maintenance are to be performed, considering, as usual, object semantics. The mechanisms designated may be either taken from system libraries or supplied by the programmer.

3. Action-Based Programming for Embedded Systems

The work performed for this task was based on the idea that embedded systems include *irreversible operations*, that is, operations that interact with the physical environment. The performance of such operations appears incompatible with the concept of an atomic action, since atomic actions rely on rollback (or more generally, backward recovery) to restore their initial state in the case of a failure. The work we have done generalizes the atomicity concept by integrating forward and backward recovery, thus removing the incompatibility between the (generalized) atomicity concept and irreversible operations.

The solutions developed involve software recovery techniques presented within the framework of action-based programming. The recovery techniques described in the handbook represent a synthesis of exception handling and action-based programming. Exception handlers are associated with individual units of work (actions) rather than with procedures or objects. The exception handlers have access to system services not otherwise accessible in a program. These services are used to achieve appropriate forward recovery. To emphasize the nature of these enhancements, exception handlers are termed *recovery handlers.*

A more general unanticipated result of our work is that the approach we present can be used not only to increase the fault tolerance of a software system, but also to simplify management and maintenance of the system. For example, if actions are robust, it will be possible to bring an individual machine down for maintenance without extensive coordination. A robust action will abort when the site goes down and either restart when it comes up or make alternative arrangement in the interim. Our approach can also be used to support software maintenance and upgrades. We describe how recovery handlers can remap the code and data windows of the associated action during recovery. This mechanism provides on-line access to backup versions of software and can be used to transfer control from the old version to a new version of the code for an action.

4. Appendices

The following papers based on and related to work performed during the course of this project are included as appendices:

"Fault Tolerant Computing in Object Based Distributed Systems" by Mustaque Ahamad, Partha Dasgupta, Richard J. LeBlanc, and C. Thomas Wilkes. From the Proceedings of the Sixth Symposium on Reliability in Distributed Software and Database Systems (March, 1987).

"Distributed Locking: A Mechanism for Constructing Highly Available Objects" by C. Thomas Wilkes and Richard J. LeBlanc. An abbreviated version of this paper will appear in the Proceedings of the Seventh Symposium on Reliability in Distributed Systems (October, 1988).

"The Clouds Distributed Operating System" by Partha Dasgupta, Richard J. LeBlanc and William F. Appelbe. From the Proceedings of the 8th International Conference on Distributed Computing Systems.

Fault Tolerant Computing in Object Based Distributed Operating Systems

Mustaque Ahamad, Partha Dasgupta,
Richard J. LeBlanc, & C. Thomas Wilkes †

School of Information and Computer Science
Georgia Institute of Technology, Atlanta, GA 30332-0280

Abstract

Replication of data has been used for enhancing its availability in the presence of failures in distributed systems. Data can be replicated with greater ease than generalized objects. We review some of the techniques used to replicate objects for resilience in distributed operating systems.

We discuss the problems associated with the replication of objects and present a scheme of replicated actions and replicated objects, using a paradigm we call PETs (parallel execution threads). The PET scheme not only exploits the high availability of replicated objects but also tolerates site failures that happen while an action is executing. We show how this scheme can be implemented in a distributed object based system, and use the *Clouds* operating system as an example testbed.

1. Introduction

A distributed system consists of many computers which are connected via communication links. The increased number of components (i.e., machines, devices and communication links) increases the chances of a failure in the system (or decreases the mean time between failures). Guarding against the effects of failures is one of the key issues in distributed computing. In this paper, we discuss approaches that provide forward progress despite the failure of some components in a distributed computing system.

† This research was partially supported by NASA under contract number NAG-1-430 and by NSF under contract number DCS-84-05020.

Authors' Address:
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332
Phone:
(404) 894 2572
Electronic Address:
{mustaq,partha,rich,wilkes} @ Gatech.edu
{akgua,allegra,hplabs,ihnp4}!
gatech!{mustaq,partha,rich,wilkes}

Our model of the distributed system is a prototype under development at Georgia Tech named *Clouds*. *Clouds* is a decentralized operating system providing location transparency, transactions, and robustness in an object based environment. In this paper, we present a review of known techniques for fault tolerance using replication. Then we discuss the salient features and architecture of *Clouds*. Finally, we present mechanisms needed for replication, probes, and parallel action threads for providing fault tolerant computing in *Clouds*. We discuss the pitfalls and the solutions to the problem of providing replication of objects having a general structure, which is more complex to achieve than replication of *flat* data (data that is accessed through read and write operations, such as files).

2. Replication Techniques for Database Systems

The use of replication to enhance availability was first studied in the area of distributed database systems, and was later adopted in the area of distributed operating systems.

2.1 Concurrency Control of Replicated Data

One of the main issues in handling of replicated data in database systems is to maintain consistency. This is achieved by concurrency control protocols. The concurrency control and recovery techniques for replicated data are summarized by Wright.[Wrig84a] He classifies these methods as *conservative (pessimistic, blocking)* and *optimistic (non-blocking)*.

Conservative Concurrency Control Methods Examples of conservative methods are voting schemes,[Giff79a, Thom79a] primary copy methods,[Ston79a] and token-passing schemes.[LeLa78a] These methods ensure consistency of the replicated data by requiring access to a special copy or a set of copies of the data. Primary copy methods allow access to a copy during a network partition only if the partition possesses the designated primary copy of the data. Token-passing schemes are an extension of primary copy methods. A token is passed among sites holding a copy of data, and the copy at the site currently holding the token is considered the primary copy. In the voting schemes, each copy of the data is assigned a (possibly different) number of votes and a partition possessing a majority of the votes for that object may access it. The conservative schemes are called *blocking* since the data is not available at a site in a partition which does not possess the primary copy (or token or majority of votes). Thus, the access must block until the partition is ended, even if a copy of the data is available in the partition. Indeed, under these schemes it is possible that *no* partition may have access to the data.

Optimistic Concurrency Control Methods The optimistic methods do not seek to ensure global consistency of replicated data during partitions.[Davi81a, Davi82a] Thus, accesses are not blocked if a replica of the data is available in the partition in question. Rather, inconsistencies in the replicas are resolved by use of backouts or compensatory actions during a merge process, once the partition is ended. It is assumed that the number of such inconsistencies will be small (hence, *optimistic*). However, tradeoffs may be made between consistency and availability. For example, the *Data-Patch* tool for designing replicated databases[Blau82a, Garc83a] assumes that, rather than strict consistency, a reasonable view of the database should be maintained to enhance availability.

3. Replication in Operating Systems

Research in database systems has been limited to consideration of flat data, and as we show later, the generalization to replication of objects having arbitrary structure leads to many problems.

These include the mechanisms used for the copying of state among replicas and having to deal with multiple instances of a single operation invocation (or a procedure call). The distributed operating systems that provide replication of objects or abstract data types include the Eden system developed at the University of Washington, the ISIS system at Cornell, and the Circus replicated call facility built on top of Unix. The replication of abstract data types has also been studied by Herlihy.

Eden The Eden system[Alme83a] has been operational at the University of Washington since April 1983. Support for replication in the Eden system has been studied at both the kernel level and the object level. The kernel level implementation of replication support is called the *Replect* approach (for replicated Ejects, or Eden objects), while the object level implementation is called *R2D2* (for Replicated Resource Distributed Database). Both implementations use quorum consensus for concurrency control.

ISIS The ISIS system developed at Cornell[Birm84a, Birm85a] supports *k-resilient* objects (operations on such an object survives up to k site failures) by means of checkpoints. This system provides both availability and *forward progress;* that is, even after up to *k* site failures, enough information is available at the remaining sites possessing the object replicas that work started at the failed sites can continue at these remaining sites. This is accomplished through a *coordinator-cohort* scheme, where a transaction executes at the coordinator site and the updates it performs on any objects are propagated to the cohort replicas. one replica acts as master during a transaction to coordinate updates at the other slave replicas (cohorts). The choice of which replica acts as coordinator may differ from transaction to transaction. The object state is copied from the coordinator to the cohorts. We call this method of state propagation *cloning*. This operation has been described as propagating a checkpoint of the entire coordinator,[Birm84a] or, in a more recent paper, as propagating the most recent version in a version stack.[Birm85a]

In ISIS, a transaction is not aborted when a machine on which its coordinator is running fails (transactions are usually aborted only when a deadlock situation arises). Rather, the transaction is resumed at a cohort from the latest checkpoint. This cohort becomes the new coordinator. Operations which the coordinator had executed after the latest checkpoint took place must be re-executed at the new coordinator.

Circus Cooper has investigated a *replicated procedure call* mechanism called Circus which was implemented in UNIX.[Coop85a] In Cooper's scheme, although replicas of a module have no knowledge of each other, they are bound (via run-time support) into a server called a *troupe* which may be accessed by clients. (The client knows that the server is replicated.) A module in Circus may have arbitrary structure, containing references to other modules. However, the module is currently required to be deterministic. His scheme uses *idemexecution* (operation execution at each replica) for state propagation. When a troupe accesses an external troupe, results of operations on modules of the server troupe are retained by the callees. These results are associated with *call sequence numbers,* and are returned when subsequent calls by the replicas of the caller troupe with the same sequence numbers are encountered. This avoids the inconsistencies that can be caused by multiple executions of the same call.

Herlihy's Work Herlihy[Herl84a] uses semantic knowledge of arbitrary abstract data types (objects) to enhance the quorum consensus concurrency control method. Analysis of the algebraic structure of data types is used in the choice of appropriate intersections of voting quorums.

4. Basics of the Clouds Operating System

Clouds is a distributed operating system that supports *objects* and *actions*. The rest of this paper deals with a set of techniques that implement generalized replicated objects in the framework of the *Clouds* operating system. We discuss the salient features of *Clouds* in this section. For a more detailed description, the reader is referred to [Dasg85a].

Figure 1 shows the hardware configuration of the *Clouds* prototype. The *Clouds* operating system provides support for the following facilities:

Distribution *Clouds* has been designed with loosely coupled distribution in mind. The hardware architecture consists of a set of general purpose machines connected by an Ethernet. The software architecture is a set of cooperating sub-kernels, which implement a monolithic view of the distributed system.

Object Based All system components, services, user data, and code are encapsulated in objects. The object structure is shown in figure 2. The *Clouds* universe is a set of objects (and nothing but objects). An object is a permanent entity, occupying its own virtual address space. Processes can weave in and out of objects through entry points defined in the object space. The only way to access data in an object is to use a process that executes the code in the object via an entry point.

Location Independence The *Clouds* objects reside in a flat, system-wide name space (the system name space is flat, the user name space need not be). There are no machine boundaries. Any process that has access to an object can invoke an operation defined by the object. This creates a unified view of the system as one large computing environment consisting of objects, even though each site in the system maintains a high degree of autonomy.

Synchronization Objects are sharable, that is several processes can invoke the object concurrently. This can pose synchronization problems. *Clouds* implements an automatic as well as custom synchronization support for concurrent access to objects. (Automatic synchronization uses two-phase locking, using read and write locks. Custom synchronization is the responsibility of the object programmer.)

Actions To prevent inconsistency in the data stored in objects, *Clouds* supports top-level and nested actions. Two-phase commit it used to ensure that all objects touched by an action are either updated successfully on a commit or are rolled back in case of explicit aborts or failures. The action management system tracks the progress of actions and maintains information about objects touched by the action and its subactions. The action management system uses the mechanisms provided by the recovery management component of the *Clouds* kernel, for performing the commit or abort operations when a action terminates or fails. Recovery management is implemented as part of the storage manager.

Clouds is designed to support a high degree of *fault-tolerance*. The mechanisms that provide this support are the topic of discussion in the rest of this paper. The following section discusses the approaches.

5. Fault Tolerance

One of the basic goals that motivated the design of *Clouds* was achieving fault tolerance. Several of the mechanisms currently supported by *Clouds* are geared to this end. Thus, we believe it is an ideal environment for building a fault tolerant system. We review some of the low level details that provide such support.

1. The object invocation strategy was designed for fault tolerant systems. When a process invokes an object (using its capability), and the object is not available locally, a global search-and-invoke is initiated.[Spaf86a] This will successfully invoke the object if it is *reachable*. Failure of any site not containing the object will not affect the invocation. The invocation will also find the object, if reachable, irrespective of where it is located, even if it was moved around in the recent past. Migration, failure, creation and deletion of objects etc. do not adversely affect the invocation mechanism.

2. All disk systems are dual-ported (or if possible, multi-ported). If a site fails, the disks belonging to the failed site are re-assigned to other working sites. Due to the location search-and-invoke mechanism, this switch can be done on the fly, and the objects that were made inaccessible due to the failure become accessible.

3. Users are not hard-wired to the sites, but are attached to logical sites through a front-end Ethernet (multiple Ethernets may be used for higher reliability, without changing our algorithms or architecture). If the site the user is attached to fails, some other site takes over and the user still has access to the system.

4. The system maintains consistency of all data (objects) in the system by using the atomic properties of actions (or transactions). *Clouds* implements nested atomic actions. This is the function of the action management system, which uses the synchronization and recovery provided at the kernel level. The commit and abort primitives are implemented in the kernel,[Pitt86a] and the action manager implements the policies. Nested actions have semantics similar to that defined in[Moss81a] and are used to firewall failed subactions.

All these mechanisms provide a certain degree of fault tolerance, that is, the system is not affected adversely by failures. Some actions are aborted, but the system as a whole continues functioning in spite of site failures. Though dual porting of disks does simulate some replication (that is, if a site fails, the data stored at the site is still available through an alternate path), this mechanism is not completely general because it can not tolerate media crashes. Also, actions executing on the failed site are forced to abort.

The action management scheme provides backward recovery and ensures that all data in the system remain consistent in spite of failures. However, this does not guarantee forward progress, as failures cause actions to abort. Fault tolerance should imply some guarantee of forward progress, that is an action should be able to continue in spite of a certain number of failures. We now discuss strategies that guarantee forward progress despite failures.

5.1 Primary/Backup Actions and Probes

One of the methods that allows fault tolerant behavior is the use of the primary/backup paradigm for actions. This paradigm is also used for fault-tolerant scheduler, monitor, and other subsystems requiring some degree of reliability.[McKe84a, Dasg86a] In this scheme, a fault-tolerant action is really two actions, one being the primary, which does the work, and the other being a backup, which is a hot standby. The primary and backup use probes to ensure both are up. If the primary fails, the backup takes over (and creates a new backup). If the backup fails, the primary creates a new backup.

The primary/backup system can be implemented using the *Clouds* probe management system. In *Clouds*, a probe can be sent from a process to another process or an object. The probe causes a quick return of status information of the recipient. Probes work synchronously, and use high priority messages and non-blocking routines so that the response time is practically guaranteed. This allows use of timeouts to check for reachability or liveness.

If a particular object is unavailable due to some failed component (even though we have dual ported disks), both the primary and the backup actions are doomed to fail. Thus the primary/backup scheme has to be augmented with increased availability of objects. Replication is the well known technique for achieving higher availability of data.

5.2 Replication of Objects

Maintaining consistency of replicated data (i.e., files) is simpler than maintaining consistency of replicated objects because only the read and write operations are provided to access data. Objects, on the other hand, are accessed through operations defined in the objects, which in turn can call operations defined in other objects. This gives rise to the following problems:

1. Due to non-determinism, the same operation invoked on two identical copies of an object may produce different results. Thus non-determinism cannot be handled in the Circus system, because it uses idemexecution.
2. Due to the nested nature of the objects, two copies of a replicated object may make a call to a non-replicated object, causing two calls where there should have been one. This can happen in the ISIS system when the coordinator crashes and some other site becomes the coordinator. In Circus this happens when the caller object is replicated.
3. Maintaining varying degrees of replication of objects produces a fan-in fan-out problem that is not easy to handle. Also, the naming scheme for replicated objects presents a non-trivial problem.

The generality of the abstract object structure supported by *Clouds* poses problems for replication methods which are not presented by objects of lesser generality. The problem lies in the possibility of the arbitrarily complex *logical* nesting of *Clouds* objects. Although *Clouds* objects may not be *physically* nested (that is, one object may not physically contain another object), an object may contain a capability to another object. If object *A* creates another object *B*, and retains sole access to *B*'s capability (by refraining from passing the capability to other objects and also not registering the capability with the object filing system [OFS]), we say that object *B* is *internal to* object *A*. The internal object *B* may be regarded as being *logically* nested in object *A*. If, on the other hand, object *A* passes *B*'s capability to some object not internal to *A*, or if *A* registers *B*'s capability with the OFS, we say that *B* is *external* to *A*. An external object is potentially accessible to objects that may not be internal to the object's creator.

Problems arise with replication schemes when internal and external objects are mixed together in the same structure, i.e., when an object may contain capabilities to both internal and extern objects. These problems are associated with the method which is used to propagate the state of a replicated object among its replicas. External objects cause problems when idemexecution is used to propagate state changes among replicas. If the replicated object invokes an operation on an external object (e.g., a print queue server), then under idemexecution, that operation will be executed by each replica. If the operation being performed on the external object is not idempotent, this can cause serious problems (e.g., multiple submissions of a job to the print queue). Also, trouble may arise when idemexecution is used if the operation on the external object is non-deterministic (for instance, random number generation, or disk block allocation among multiple concurrent processes).

On the other hand, internal objects cause problems when cloning is used to propagate state. For example, assume that each replica of an object creates a set of internal objects. Then, when an operation is performed on one of the replicas, its state under cloning is copied to each of the other replicas. However, since the capabilities to the internal objects of the replicas are contained in their states, each replica now contains capabilities to the internal objects of the replica at which the operation was actually executed. Thus, the information about the internal objects of the other replicas is lost.

6. Replication Mechanisms

6.1 Replicated Actions

We have developed a scheme called *replicated actions*. Each replicated action runs as a nested action and has its own thread of execution. Each thread of control is called a *Parallel Execution Thread* or PET. The degree of the replicated action is the number of PETs that comprise the action. The degree is determined statically at the the time the top level action is created. If all objects touched by the action are replicated *k* times and the degree of the replicated action is also *k*, we can have each PET executing on a different copy of the object.

Briefly, the PET scheme sets up several parallel, independent actions, performing the same task, using a possibly different set of replicas of the objects in question. These actions follow different execution paths, on different sites, but only one of them is allowed to commit. The scheme is depicted in Figure 3, and its implementation details are presented in Section 6.4.

The PET scheme for replicated objects has several advantages. Firstly, up to k-1 transient failures (in a PET scheme with k threads), are automatically handled because the remaining PETs will commit the action. This contrasts with the ISIS scheme in which one of the sites having a replica has to detect the failure of the coordinator and assume responsibility for the execution of the action. However it is possible for an action in ISIS to commit while all the PETs may abort in our scheme. The possibility of this happening is considerably reduced as the degree of the PETs are increased. Thus this scheme presents a trade off between computation and replication (overhead) and the degree of fault tolerance.

A replica of an object that is replicated k times can receive multiple calls (as in ISIS and Circus) when the PET degree is more than k. Thus a replica has to retain results to avoid executing the same call operation again. However a caller will not receive multiple results as in Circus and we do not have to collate the returned results. Also since only a single PET is allowed to commit, cloning is used for state copying and non-deterministic operations do not cause inconsistent state in the replicas. The problem of internal (or nested) objects is solved by a modification of the capability (naming) scheme, which is described below.

6.2 Naming Replicated Objects

Replicated objects and actions provide support for guaranteeing forward progress when system components fail. This introduces the problem of naming replicated objects. In *Clouds*, the system uses a capability based naming scheme. A capability is a system name which uniquely identifies one object in the distributed system. Under this scheme, a *k*-replicated object is named by *k* different capabilities. This makes naming considerably more difficult, and since capabilities are stored within an object, state copying via cloning causes the problems described earlier.

To solve this we propose a minor modification to the capability scheme. When replication is supported by the kernel, at the user level, all copies of the replicated object have the same capability, and thus one capability refers to a set of objects. A flag in the capability tells the kernel that the capability points to a set of replicas of the object.

The kernel can then append a *copy number* to generate unique references to the objects. The kernel uses the <capability:copy-number> pair to invoke operations. Thus the kernel can choose to invoke the appropriate copy (or several copies) depending upon the replication algorithms used to resolve an invocation on a replicated object.

Since all references to the object, as far as the program is concerned, are still made through a unique capability, which points to all the copies, any naming problems at the user level disappear (when replication is supported by the kernel). Constructing the <capability:copy-number> pair can be effectively handled at the kernel level, using one of several techniques. (For example, the copy number 1 is always valid, and this copy, as well as other copies, contain information about the total number of copies, and thus all copies are accessed by the range 1..max.) This scheme is depicted in Figure 4.

6.3 Invocation of Replicated Objects

The invocation scheme for replicated objects has to follow the scheme outlined above. The kernel interface handles invocation as follows. For simplicity, in this section we will assume al he actions have only one thread of control (1-PET). We will generalize the scheme in the next secti 1.

A process executing on behalf of an action requests the invocation of an operation defined by n object. The kernel examines the capability and detects whether the object is replicated or not. If it is not replicated, the invocation proceeds as a normal *Clouds* invocation. If the capability points to a replicated object, the kernel has to choose one of the replicas. If a local copy of the object is available, the kernel invokes the local copy, else it tries to invoke any one copy, by appending the copy number and sending out an invocation request on the broadcast medium. Typically, the kernel chooses copy number 1, and if that fails it tries subsequent copies. This sequential searching is not necessary, as the kernel can use previous history to decide which replica to use.

Once a replica is used for an action, the kernel takes note of that, and stores it with the action id, and all later invocations are directed to that replica. Thus only a single replica of each replicated object is used to execute one action. The other replicas are not touched, until the action decides to commit. When an action commits, the replica it touched is copied to all other replicas. This is done by copy requests from the action management systems to all the replicas (using the copy number scheme). All accessible replicas are updated and their version numbers updated. (Note that if the source object has a copy number lower than a replica, the action has to be aborted.) The version copying strategy is shown in Figure 5. The version numbers are also used to bring failed sites up-to-date on startup. On startup, all replicas at the site having version numbers less than the highest version number on the network are reinstated.

6.4 Handling PETs

The above scheme using 1-PET execution is prone to failures in certain cases. These include cases where a replica becomes unavailable after it has been invoked, the replica invoked was not up-to-date and when the site coordinating the action fails.

The N-PET (N>1) case decreases the chances of transaction abort due to the transient failures described in the earlier paragraph. All the separate PETs have different co-ordinating sites and execute independently.

When the first thread invokes a replicated object, the invocation proceeds as above, that is a replica is chosen to service the action. The second thread also proceeds similarly, but a different replica is chosen. The replica choice does not have to be different, but the reliability increases if they are, so we use a random choice scheme. Note that the same object is chosen (as there is no choice) if the object is not replicated. Multiple invocations of the same object, due to multiple threads of control are handled by a collator. The commit phase is however different.

In this scheme, ONLY one PET can be allowed to commit. If more than one PET reaches commit point, each PET issues a pre-commit, which checks if all the primary copies it touched are still available. If any thing is not reachable, the PET aborts. Of the remaining PETs any one has to be chosen to commit (In fact if all of them are allowed to proceed, they will overwrite each others results and may cause deadlocks during commit time.) The co-ordinating site with the highest site number wins the match and commits the PET that was associated with the site. The commit causes the replicas touched by this PET to be copied to all other replicas. The co-ordinating sites that lost the commit war, do not abort the PETs, but wait for the commit of the winner to be over. If the commit fails the co-ordinator with the next highest site number attempts the commit. (Note that the previous commit could have attempted to overwrite the replicas touched by this PET, but the pre-commit causes a special copy of all the replicas to be retained, and this copy is used for the commit.)

Transient failures cause failed PETs, but the chances of all PETs failing decreases as the number of PETs is increased. Also, failures during commit are taken care of, by the other PETs. Of course it is possible for all the PETs to abort, but the chances of this happening decrease as the replication degree and the PET degree is increased.

7. Concluding Remarks

There are two major contributions of this research.

1. The object replication scheme is not as straightforward as data replication. The capability scheme allows reference to a set of objects and the cloning technique ensures correct execution in spite of generalized and nested objects, as well as non-deterministic objects.

2. Replication enhances availability, that is, actions can be run on a system that has some sites or data missing due to failures. Handling transient failures are not possible in most replicated schemes, that is, if an action touches an object, and the object later becomes inaccessible, before the action commits, the action has to abort. Also, once an action has visited a site, the failure of that site before the action commits can lead to action failure. The PET scheme allows the action to proceed, with high probability of success, in a unreliable environment, where sites fail and restart during the execution time of the action.

We are currently involved with designing the lower level algorithms and modifying the *Clouds* action management scheme to implement the PET method of providing fault tolerance in the *Clouds* operating system. This involves the implementation of the collators, the kernel primitives to choose the appropriate replicas, the mechanisms that ensure distinct PETS choose distinct replicas and so on. Once the implementation is complete, we will be able to experimentally study the reliability of this approach.

REFERENCES

[Alme83a] Almes, G. T., A. P. Black, E. D. Lazowska, and J. D. Noe. "The Eden System: A Technical Review." TECHNICAL REPORT 83-10-05, University of Washington Department of Computer Science, October 1983.

[Birm84a] Birman, K. P., T. A. Joseph, T. Raeuchle, and A. El-Abbadi. "Implementing Fault-Tolerant Distributed Objects." *PROCEEDINGS OF THE FOURTH SYMPOSIUM ON RELIABILITY IN DISTRIBUTED SOFTWARE AND DATABASE SYSTEMS*, Silver Spring, MD (October 1984): 124-133.

[Birm85a] Birman, K. P. "Replication and Fault-Tolerance in the ISIS System." *PROCEEDINGS OF THE TENTH SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES* (ACM SIGOPS), Orcas Island, Washington (December 1985). (Also released as technical report TR 85-668.)

[Blau82a] Blaustein, B., R. M. Chilenskas, H. Garcia-Molina, D. R. Ries, and T. Allen. "Partition Recovery Using Semantic Knowledge." (TECHNICAL REPORT), Computer Corporation of America, Cambridge, MA, November 1982.

[Coop85a] Cooper, E. "Replicated Distributed Programs." *PROCEEDINGS OF THE TENTH SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES* (ACM SIGOPS), Orcas Island, WA (December 1985): 63-78. (Available as *Operating Systems Review* **19**, no. 5.)

[Dasg85a] Dasgupta, P., R. LeBlanc, and E. Spafford. "The Clouds Project: Design and Implementation of a Fault-Tolerant Distributed Operating System." TECHNICAL REPORT GIT-ICS-85/29, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1985.

[Dasg86a] Dasgupta, P. "A Probe-Based Monitoring Scheme for an Object-Oriented Distributed Operating System." *PROCEEDINGS OF THE CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS* (ACM SIGPLAN), Portland, OR (Sept. 1986): 57-66. (Also available as Technical Report GIT-ICS-86/05.)

[Davi81a] Davidson, S. and H. Garcia-Molina. "Protocols for Partitioned Distributed Database Systems." *PROCEEDINGS OF THE SYMPOSIUM ON RELIABILITY IN DISTRIBUTED SOFTWARE AND DATABASE SYSTEMS*, Pittsburgh, PA (July 1981).

[Davi82a] Davidson, S. "An Optimistic Protocol for Partitioned Distributed Database Systems." PH.D. DISS., Department of Electrical Engineering and Computer

Science, Princeton University, 1982.

[Garc83a] Garcia-Molina, H., T. Allen, B. Blaustein, R. M. Chilenskas, and D. R. Ries. "Data-Patch: Integrating Inconsistent Copies of a Database after a Partition." *PROCEEDINGS OF THE THIRD SYMPOSIUM ON RELIABILITY IN DISTRIBUTED SOFTWARE AND DATABASE SYSTEMS*, Clearwater Beach, FL (October 1983).

[Giff79a] Gifford, D. K. "Weighted Voting for Replicated Data." *PROCEEDINGS OF THE SEVENTH SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES* (ACM SIGOPS), Pacific Grove, CA (December 1979).

[Herl84a] Herlihy, M. "Replication Methods for Abstract Data Types." PH.D. DISS., Laboratory for Computer Science, Massachussetts Institute of Technology, Cambridge, MA, May 1984. (Also released as Technical Report MIT/LCS/TR-319.)

[LeLa78a] LeLann, G. "Algorithms for Distributed Data-Sharing Systems Which Use Tickets." *PROCEEDINGS OF THE THIRD BERKELEY WORKSHOP ON DISTRIBUTED DATA MANAGEMENT AND COMPUTER NETWORKS*, Berkeley, CA (August 1978).

[McKe84a] McKendry, M. S. "Fault-Tolerant Scheduling Mechanisms." (UNPUBLISHED TECHNICAL REPORT), School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, May 1984. (Draft only.)

[Moss81a] Moss, J. "Nested Transactions: An Approach to Reliable Distributed Computing." TECHNICAL REPORT MIT/LCS/TR-260, MIT Laboratory for Computer Science, 1981.

[Pitt86a] Pitts, D. V. "Storage Management for a Reliable Decentralized Operating System." PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (Also released as Technical Report GIT-ICS-86/21.)

[Spaf86a] Spafford, E. H. "Kernel Structures for a Distributed Operating System." PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (Also released as technical report GIT-ICS-86/16..)

[Ston79a] Stonebreaker, M. "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES." *TRANSACTIONS ON SOFTWARE ENGINEERING* (IEEE) **5**, no. 3 (May 1979).

[Thom79a] Thomas, R. H. "A Majority Consensus Approach to Concurrency Control for Multiple-Copy Databases." *TRANSACTIONS ON DATABASE SYSTEMS* (ACM) **4**, no. 2 (June 1979).

[Wrig84a] Wright, D. D. "Managing Distributed Databases in Partitioned Networks." PH.D. DISS., Department of Computer Science, Cornell University, Ithaca, NY, January 1984. (Also available as Cornell University Technical Report 83-572.)

Distributed Locking: A Mechanism for Constructing Highly Available Objects

C. Thomas Wilkes[*] Richard J. LeBlanc, Jr.[†]
University of Lowell Georgia Institute of Technology

April 4, 1988

Abstract

Distributed Locking refers to a methodology for constructing replicated objects from single-site implementations in an action-based object-oriented system such as the *Clouds* project. It also refers to the mechanism provided to support this methodology in Clouds. This mechanism assumes no particular *policy* for control of replica concurrency and consistency; rather, it provides primitives with which a wide range of policies may be supported. Also, by use of extensions to the *Aeolus* systems programming language supporting *replication events*, the specification of the availability properties of an object is abstracted from the object implementation. Thus, a replicated object may be constructed from a single-site implementation, or changes made in the policies used for control of a replicated object, with little or no change to the object implementation. Examples of the specification and use of the quorum consensus replication control policy using the Distributed Locking primitives are described.

[*]This work was performed while the author was with the School of ICS at Georgia Tech. Author's present address: Department of Computer Science, University of Lowell, One University Avenue, Lowell, MA 01854. Internet: wilkes@hawk.ulowell.edu

[†]Author's address: School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA 30332-0280. Internet: rich%lynx@gatech.edu

Distributed Locking: A Mechanism for Constructing Highly Available Objects

1. Introduction

Among the benefits claimed for distributed computing are improvements in system fault tolerance and reliability, and increased availability of data and services. The *Clouds* project at Georgia Tech is one of a number of recent proposals in which reliability in a distributed system is based on the use of *atomic actions*, a generalization of the transaction concept of distributed databases. As part of the Clouds project, we have designed and implemented a high-level language providing access to the synchronization and recovery features of the Clouds system; this language is being used to implement those levels of the Clouds system above the kernel level. It also provides a framework within which to study programming methodologies suitable for systems based on the action concept, such as Clouds. Among the properties needed by systems data structures, the design of which must be addressed by such methodologies, are *resilience*—survivability and consistency of the data despite crashes and other faults; and *availability*—increased possibility of access to data despite network partitions or failures of some sites in a multicomputer system. Together with a mechanism that ensures *forward progress*—continued execution of jobs despite failures, these properties provide *fault tolerance* in the system.

In tnis paper, we describe some of the results of a study of methods of achieving fault tolerance in the Clouds system, in particular achieving increased availability of objects in Clouds. The remainder of this introduction presents the problems explored by this work. Section 2 describes the model of distributed computation in which the problems posed by the research were examined (the Clouds system) and the tools which were used to address these problems (the Aeolus[1] programming language). In Section 3, we present a methodology for achieving available services by conversion of resilient single-site implementations into replicated implementations. A mechanism with which we proposes to support this methodology, called *Distributed Locking*, is also described in Section 3. In Section 4, we describe a linguistic feature for the specification of the availability properties of an object replicated via Distributed Locking. The language runtime support features (primitives) required to support Distributed Locking, as well as operating system support needed to support these features, are presented in Section 5. In Section 6, previous work in database systems is presented as well as work in the operating system area that is relevant to the author's research. Finally, the conclusions which we have drawn from this research are summarized in Section 7, as are plans for future extensions of this work.

The work described in this paper is, in general, concerned with situations in which sites fail by halting, that is, *fail-stop* failures [Schl83a]; in particular, malicious activity by failed sites (so-called *Byzantine* failures) are not considered here.

1.1 The Need for Availability

Even if a computation is distributed, it is subject to a single point of failure if any of the data objects involved in that computation exist at only a single node. The provision of resilience alone cannot eliminate the problems caused by site or network failures; although inconsistencies introduced by such failures have been abolished, any objects existing only at a failed site are unavailable for the duration of the failure, and thus no computation may proceed which requires

1. Aeolus was the king of the winds in Greek mythology.

those objects. A method for eliminating these bottlenecks is *data replication*, that is, the maintenance of copies of an object at multiple sites.

The use of replication introduces the problem of maintaining the *consistency* of the individual replicas when operations are executed on them. A common requirement for consistency is that the replicated object maintain *single-copy semantics*, that is, that the state of each replica be consistent with that which would have been obtained had the object existed only at a single site and had the same sequence of operations been applied to it. This is achieved by a combination of a mechanism for controlling concurrency among the replicas, and of a mechanism for copying the state obtained by an operation execution among the replicas.

These mechanisms have been the subject of much study, both in the areas of database systems and of operating systems. Indeed, it has been found that single-copy semantics is too stringent a requirement in some applications. (See [Wilk87a] for a discussion of previous work in this area.) However, most previous work on such mechanisms has been concerned with "flat" data, such as files. The unique problems posed for these mechanisms by the object construct used in systems such as Clouds are discussed in the following section; in so doing, we also introduce some terminology used in the remainder of this paper.

1.1.1 Problems of Replication in Object-Based Systems In the course of research on methods of achieving availability in object-based systems such as Clouds, we have found that the generality of the abstract object structure supported by Clouds poses problems for replication methods which are not presented by a less general, flat object structure (for instance, files or queues).

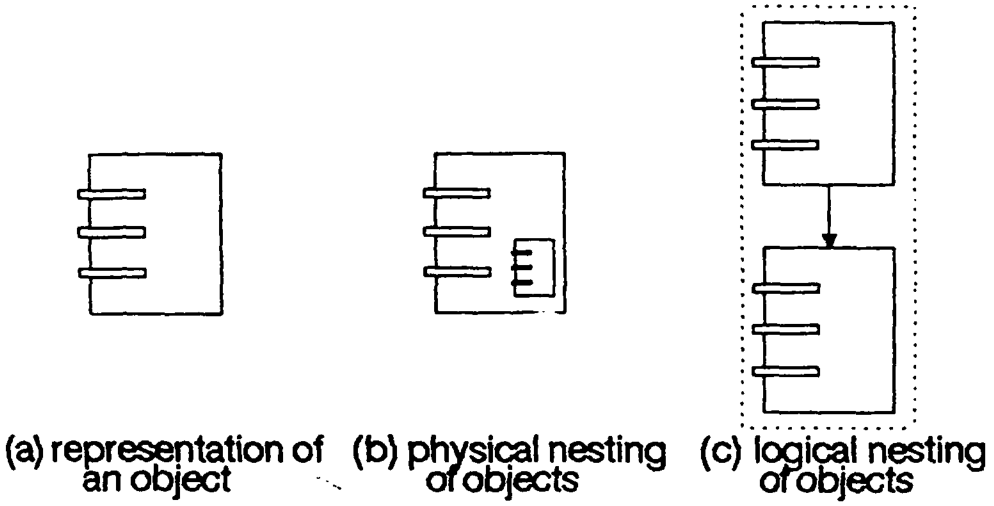
(a) representation of an object
(b) physical nesting of objects
(c) logical nesting of objects

Figure 1. Pictorial Representation of Object Nesting

The problem lies in the possibility of the arbitrarily complex *logical* nesting of Clouds objects. Although Clouds objects may not be *physically* nested (that is, one object may not physically contain another object), an object may contain a capability to another object. If an object *A* creates another object *B*, and retains sole access to *B*'s capability (by refraining from passing the capability to other objects, either explicitly or through an intermediary such as an object directory service), object *B* is said to be *internal to* object *A*. The internal object *B* may be regarded as being *logically* nested in object *A*. (A pictorial representation of physical and logical nesting is shown in Figure 1.) If, on the other hand, object *A* passes *B*'s capability to some object not internal to *A*, or if *A* registers *B*'s capability with an object directory service, *B* is said to be an *external* object; an external object is potentially accessible by objects not internal to the object

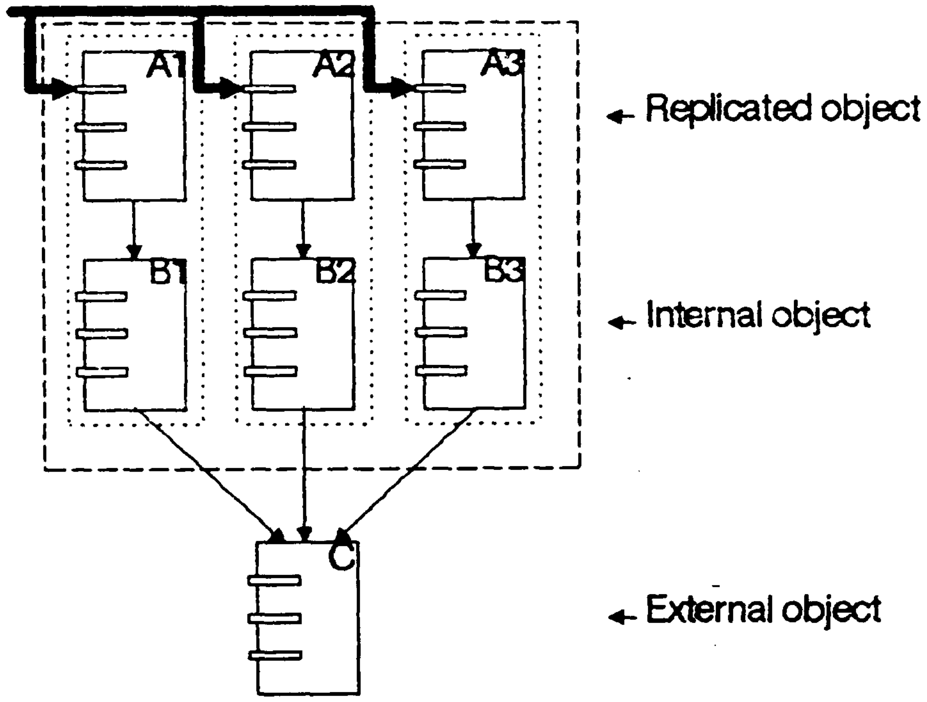

Figure 2. Replicated Object with Internal and External Object References

which created the external object.

Problems arise with replication schemes when internal and external objects are mixed together in the same structure, *i.e.*, when an object may contain capabilities to both internal and external objects. (An example of such an object is represented in Figure 2.) These problems are associated with the method which is used to propagate the state of a replicated object among its replicas. One such method is to execute **at each replica** the computation from which the desired state results; this scheme is called ***idemexecution***. Another method is to execute the computation at one replica, and then copy the state of that replica to the other replicas; this scheme is called ***cloning***. (Representations of the idemexecution and the cloning methods are shown in Figure 3.) Note that the scheme which is used to ensure that the replicas maintain consistent states (*e.g.*, quorum consensus) is not involved in these problems, and is considered separately in this investigation.

External objects cause problems when idemexecution is used to propagate state among replicas. If the replicated object performs some operation on an external object (*e.g.*, a print queue server), then—under idemexecution—that operation will be repeated by each replica. If the operation being performed on the external object is not idempotent, this can cause serious problems (*e.g.*, multiple submissions of a job to the print queue). Also, trouble may arise due to idemexecution if the operation on the external object is non-deterministic (for instance, random number generation, or disk block allocation among multiple concurrent processes).

On the other hand, internal objects cause problems when cloning is used to propagate state. For example, assume that each replica of an object creates a set of internal objects. Then, when an operation is performed on one of the replicas, its state—under cloning—is copied to each of the other replicas. However, the capabilities to the internal objects of the replicas are contained in their states; thus, each replica now contains capabilities to the internal objects of that replica on which the operation was actually performed, and the information about the internal objects of the

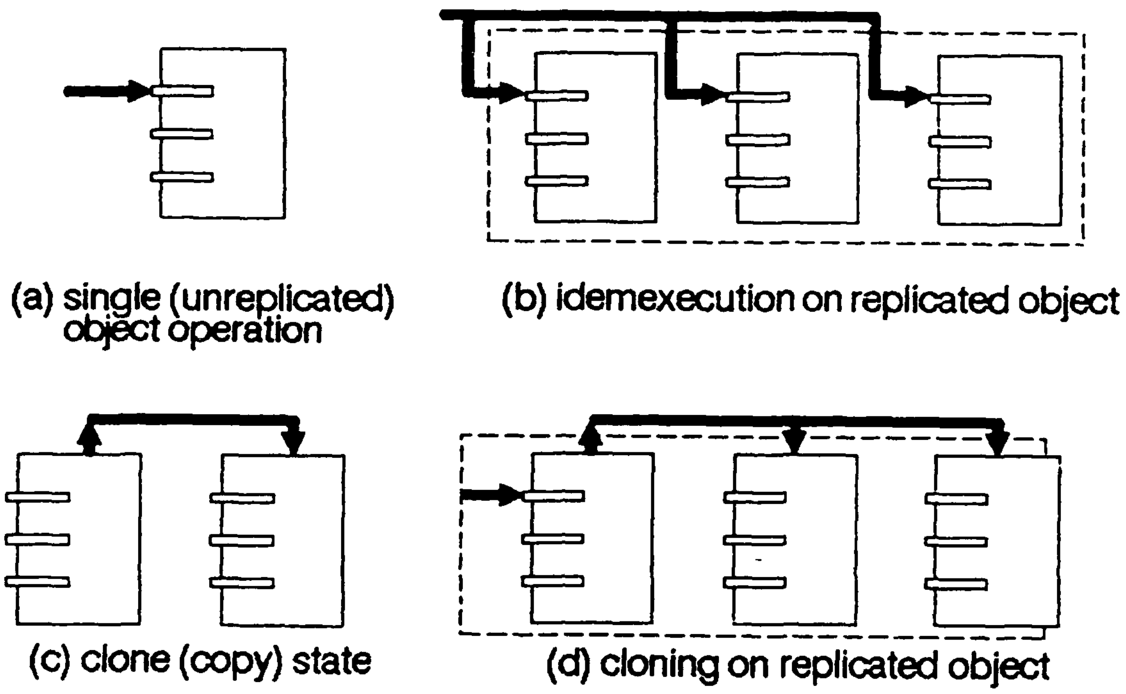
(a) single (unreplicated) object operation
(b) idemexecution on replicated object
(c) clone (copy) state
(d) cloning on replicated object

Figure 3. Replicated State-Copying Methods

other replicas is lost. This problem is illustrated in Figure 4. In Figure 4 (a), each replica has a capability to its individual internal object. In Figure 4 (b), an operation execution has taken place at the leftmost replica in the figure, and its state has been cloned to the other two replicas; the states of the other replicas now contain capabilities to the internal object of the leftmost replica rather than to their own internal objects.

1.2 The Need for Distributed Locking

In recent years, several researchers have presented algorithms that have explored the feasibility of trading consistency for availability in specific applications, or have taken advantage of semantic knowledge of typed objects to increase resilience or availability of these objects. (This related work is described in Section 6.) It has become clear from this research that, in certain applications, the ability to exploit trade-offs between consistency and availability, and to make use of the semantic knowledge of objects towards these goals, is not only feasible but desirable. It thus seemed inadvisable to limit the user to any pre-specified algorithm; none seemed sufficient to handle all of the potential applications. Accordingly, the focus of the research presented here changed to the question of how the various algorithms and techniques might be supported in the Clouds system in a general and efficient manner.

Features to support programming for resilience were introduced into the Aeolus testbed language at a relatively early stage, as these model closely the mechanisms provided by the Clouds kernel; these features are described in Section 2. Features to support programming for availability, on the other hand, were designed at a relatively late stage of this research. Our first attempts to program available objects in Aeolus soon convinced us, due to their *ad hoc* nature, of the desirability of linguistic support in this area. In these early attempts the manual addition of support for replication to an object originally designed as a single-site implementation was distressingly inelegant; an example of the result of this strategy is supplied elsewhere [Wilk87a]. This experience suggested that a proper goal would be automation (to whatever extent possible)

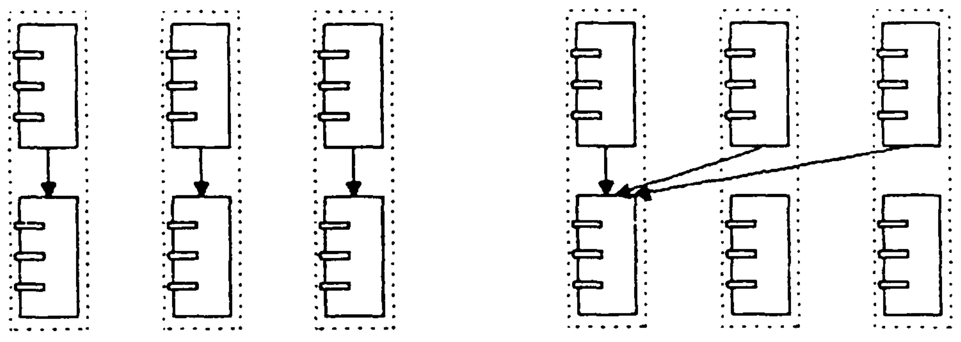

Figure 4. State Cloning with Internal Objects

of the process of deriving a replicated implementation of an object from its single-site implementation. The resulting *Distributed Locking* mechanism, described in Section 3, provides support for the control of concurrency and state-copying among replicas of an object while making no assumptions about the policies used for this control. The abundance of replication-control algorithms that has appeared in the literature in recent years, often taking advantage of the semantics of a particular application, makes it clear that limiting support to any particular policy would be undesirable. Rather, primitives are provided to support programming of custom replication-control policies which may take advantage of semantic knowledge of objects; however, options are provided for the automatic use of one of several common replication-control algorithms, if desired. This is in accord with the philosophy of the Clouds system as demonstrated by its treatment of the issues of synchronization and recovery. The linguistic support added to Aeolus to aid the programmer in the specification of the availability properties of an object is described in Section 4.

2. The Aeolus/Clouds Model

In this section, we provide an overview of the model of distributed computation embodied in the Aeolus/Clouds system. The background of the Clouds distributed operating system project, as well as the major concepts and facilities presented by the Clouds system, are presented here; a more complete description of the system may be found in a recent overview paper [Dasg88a]. Also, the major features of the *Aeolus* language are described briefly.

2.1 The Clouds System

The Clouds distributed operating system project has been under development at Georgia Tech since late 1981; the central concepts were developed by Allchin and McKendry in a pair of early papers [Allc82a, Allc83a], and the Clouds architecture was described in full in Allchin's dissertation [Allc83b]. The goal of the Clouds project is the implementation of a fault-tolerant distributed operating system based on the notions of *objects*, *actions*, and *processes*, to provide an environment for the construction of reliable applications on unreliable hardware. The basic approach is to exploit the redundancy available in distributed systems which consist of multiple computers connected by high-speed local area networks. Such systems are called *multicomputers* or *computer clusters*. In Clouds, the notion of an *object* may be used to represent system components, such as directories or queues. A set of changes to objects may be grouped into an *action*, which corresponds roughly to the *transaction* concept of distributed database work, providing an "all or nothing" assurance of atomic execution (a property sometimes called *failure* *atomicity*). The underlying support system ensures that, even if the actions extend across multiple machines, the changes will occur in totality or not at all. At this level, the support system, known as the *Clouds kernel*, is maintaining the *consistency* of the objects. It ensures that objects either reflect the effects of an action totally or not at all—no intermediate states are possible. This guarantee of an action's totality permits one to characterize the effects of hardware component failures: they cause actions to fail. Since a failed action is guaranteed to have had no effects on the objects with which it interacted, the action may be restarted without concern for potential inconsistencies it might have created.

Actions in Clouds go beyond the related notion of transactions in a database system. Rather than modelling all access to objects as simple reads or writes, the Clouds approach supports arbitrary operations on objects and allows a programmer to take advantage of operation semantics to increase concurrency, and thereby, performance. Through appropriate use of encapsulation, concurrent actions can be allowed to change objects without violating serializability.

A powerful feature of Clouds is the separation of the two components the traditional notion of the serializability of atomic actions, *failure atomicity* and *view atomicity*. Failure atomicity, as mentioned above, refers to the "all or nothing" property of atomic actions; view atomicity requires that the effects of an uncommitted action are not seen by other actions until commital occurs, thus avoiding the problem of "cascading aborts" of actions which have viewed intermediate states of an uncommitted action that later is aborted. This separation of the recovery and synchronization aspects of serializability allows the Clouds programmer to design objects that, while maintaining an appearance of serializability to the outside world, may violate strict serializability internally—in ways based on the programmer's knowledge of the object's semantics—in the interest of system efficiency.

Objects, actions, and processes are fundamental concepts supported by the Clouds architecture. To support these concepts, recovery and consistency are incorporated into the basic virtual memory mechanism [Pitt86a, Pitt88a]. Synchronization mechanisms to control the interactions of actions are also provided. It is with these capabilities that Clouds is meant to support the data integrity required for the implementation of reliable, distributed application programs.

The detailed design of the Clouds kernel is discussed in Spafford's dissertation [Spaf86a]. A prototype of the Clouds kernel, also described by Spafford, has been implemented on a hardware testbed consisting of VAX® 750s connected by a 10Mbps Ethernet, several dual-ported disk drives, and Sun 3 Workstations® running UNIX®—also attached to the Ethernet—that provide a user interface to the Clouds system. The Clouds kernel is implemented "on the bare machine," that is, it is not implemented on top of some other operating system such as UNIX. Thus, the features of objects, actions, and processes have been implemented in the lowest levels of the kernel, allowing use of the Clouds concepts in the construction of the operating system itself. At these lowest levels, we attempt to avoid implementing *policies*, instead providing *mechanisms* with which policies may be constructed. Some policies are embedded in subcomponents of the kernel. The storage management system [Pitt86a] implements support for action-based stable storage within the object virtual memory mechanism. The action manager [Kenl86a] controls the interaction of actions with objects, including creation, committal, and abortion of actions, a

® VAX is a registered trademark of Digital Equipment Corp.

® Sun Workstation is a registered trademark of Sun Microsystems, Inc.

® UNIX is a registered trademark of AT&T.

time-based orphan detection facility. and support for lock-based synchronization. Those kernel subcomponents implementing policy are intended to be replacable with minimal changes to the rest of the kernel. For instance, the storage management system could be replaced with another implementing log-based recovery, or the action manager changed to support timestamp-based synchronization, without fundamental changes to other kernel subcomponents.

The Clouds system above the kernel level consists of a set of fault-tolerant *servers* which provide system services (such as object filing, job scheduling, printer spooling, and the like) to application programs. (It is for the construction of this level of the Clouds system that the Aeolus programming language was designed; the kernel itself has been implemented in the C language.)

The location-transparency and resilience mechanisms provided by the Clouds architecture are used to support the operating system itself and its services. Thus, the system itself is decentralized (in the sense that the system can survive the failure of any node) and resilient. The Clouds system may be considered to consist of a set of fault-tolerant objects which in combination provide a reliable environment for applications.

2.2 The Aeolus Programming Language

In this section we provide a brief overview of the Aeolus programming language. More complete discussions of Aeolus may be found in previous publications [Wilk85a, Wilk86a, Wilk87a].

Aeolus developed from the need for an implementation language for those portions of the Clouds system above the kernel level. Aeolus has evolved with these purposes:

- to provide the power needed for systems programming without sacrificing readability or maintainability;
- to provide abstractions of the Clouds notions of objects, actions, and processes as features within the language;
- to provide access to the recoverability and synchronization features of the Clouds system; and
- to serve as a testbed for the study of programming methodologies for action-object systems such as Clouds [LeBl85a].

The intended users of Aeolus are systems programmers working on servers for the Clouds system. Clouds provides powerful features for the efficient support of resilient objects where the semantics of the objects are taken into account; it is assumed that the intended users have the necessary skills to make use of these features. Thus, although access to the automatic recovery and synchronization features of Clouds is available, we have avoided providing very-high-level features for programming resilient objects in the language, with the intention of evolving designs for such features out of experience with programming in Aeolus.

2.2.1 Support For Synchronization Aeolus provides access to the action manager's support for synchronization via a *lock* construct. An unusual aspect of Aeolus/Clouds locks is that they are associated not with the specific data being locked, but rather with values in some *domain*. Thus, an lock is obtained for a value of an object, and not on the object itself. Thus, for instance, a lock may be obtained on a file name even if that file does not yet exist. Another interesting feature of Aeolus/Clouds locks is that they provide a mechanism for the specification of arbitrary locking *modes* and arbitrary compatibilities between the different modes, thus allowing the lock to be tailored to the specific synchronization semantics of a subset of object operations. For example:

```
type file_lock is lock ( read  : [ read ],  write : [] )
        domain is string( FILE_NAME_SIZE )
```

The declaration of `file_lock` defines a lock type over the domain of strings representing filenames, in which the usual multiple reader/single writer synchronization is specified by the compatibilities among the `read` and `write` modes of the lock.

All locks obtained during execution in the environment of a nested action are retained and propagated to the immediate ancestor of that action upon committal unless explicitly released by the programmer. Locks obtained under an action are automatically released if the action aborts or successfully performs a toplevel commit. Thus, a two-phase locking protocol (2PL) is maintained, with violations to 2PL allowed (via explicit release of locks) if the programmer deems such violations acceptable. A lock is available to be granted under a nested action even if conflicting locks are held under one or more of the ancestors of that action, but not if conflicting locks are held under an action which is not an ancestor of the nested action [Allc83b]. The power of the Aeolus/Clouds lock construct in supporting user-defined synchronization lies in the specification of arbitrary locking modes, and arbitrary compatibilities between those modes, as well as the dissociation of locks from the locked variables.

2.2.2 Support for Objects The *object* construct provides support for *data abstraction* in Aeolus. A collection of related data items may be *encapsulated* within an object, which also may provide *operations* (procedures that operate) on the data. The only access to the data of an object is via these operations; thus, an object can strictly control manipulation of its encapsulated data, helping guarantee the invariants of the abstraction. The declaration of the object defines a type, called an *object type*, which may be used in the declaration of variables to hold capabilities to *instances* of that object type.

Aeolus provides a hierarchy of *object classifications* sharing a common implementation and invocation syntax which offers a trade-off of functionality and efficiency. The object classifications fall into two groups: the so-called Clouds object classifications (`autorecoverable`, `recoverable`, and `nonrecoverable`) may make use of the object management facilities and (for `autorecoverable` and `recoverable` types) the action management facilities, while the non-Clouds object classifications (`local` and `pseudo`) do not use any of the Clouds facilities for action or object management and provide data-abstraction facilities usable "locally" (without resorting to the system facilities supporting distribution of objects). On the other hand, the Clouds object classifications provide access to the support for data abstraction provided by the Clouds system when the expense of that support is warranted; the separate classifications of Clouds objects allow the programmer to specify the degree of support (and of incurred expense) required. The object classifications are described in more detail in the papers cited above; while the `autorecoverable` classification provides the paradigm most often presented by other action systems, that is, completely automatic recovery of the entire object state, the `recoverable` classification is of more interest here in that it allows the programmer to tailor object recovery based on the semantics of the object via mechanisms described below.

The global variables of an object are called collectively the object's *state*. In an object of class `recoverable`, part of the object state may be specified to be in a *recoverable area;* also, the programmer may specify an *action events part* and/or a *per-action variables part*. Recoverable areas, action events, and per-action variables are described below.

In order to allow the object to participate in its own creation and deletion, an object implementation part contains specifications of handlers for the so-called *object events*. The object events include the *init* or object initialization event, the handler for which is executed whenever an instance of the object is created by use of an allocator; the *reinit* or object reinitialization event, the handler for which is executed—if the object has registered its desire for reinitialization with the action manager—when the system is reinitialized after a crash or network partition; and the *delete* or object deletion event, the handler for which is executed when the object instance is destroyed.

An invocation of an object operation looks much like a procedure invocation, except that, outside the implementation part of the object itself, an operation name must be qualified by the name of a variable representing an instance of that object type (or, for pseudo-objects, by the name of the object type itself). Thus, for an instance of a bounded-stack type, the programmer might write

```
stack_instance @ push( elem )
```

When an object invokes one of its own operations, however, the usual procedure call syntax is used.

Invocations of pseudo-object and local object operations have semantics essentially similar to those of calls to procedures local to a compiland. The situation is different for operations declared in objects which use the Clouds object-management facilities (*i.e.*, the so-called "Clouds objects"). Invocations of operations on Clouds objects are handled by the compiler through operations on the Clouds object manager on the machine on which the invoking code is running. The Clouds object on which the operation is being invoked need not be located on the same machine as the invoking code; the object manager then makes a *remote procedure call* (RPC) to the object manager on the machine on which the called object resides. The location—local or remote—of the object being operated upon, however, need not concern the programmer, as the RPC process is transparent above the object-management level.

2.2.3 Support for Actions The action concept provides an abstraction of the idea of work in the Clouds system; an action represents a unit of work. Actions provide *failure atomicity*, that is, they display "all-or-nothing" behavior: an action either runs to completion and *commits* its results, or, if some failure prevents completion, it *aborts* and its effects are cancelled as if the action had never executed.

Support for actions in the Aeolus language is relatively low-level. At present, the methodology of programming with actions is not as well-understood as the methodology of programming with objects; thus, rather than providing high-level syntactical abstractions such as those available for object programming, Aeolus allows access to the full power and detail of the Clouds system facilities for action management. The major syntactic support provided by Aeolus for action programming is in the programming of *action events*, *recoverable areas*, *permanent* and *per-action variables*, and *action invocations*.

At several points during the execution of an action, the action interacts with the *action manager* of the Clouds system to manage the states of objects touched by that action, including writing those states to *permanent* (stable or safe) storage, and recovering previous permanent states upon failure of an action. Thus, failure atomicity may be provided by the action management system. The *action events* include:

event name	*purpose*
`BOA`	beginning of action
`toplevel_precommit`	prepare for commit of a toplevel action
`nested_precommit`	prepare for commit of a nested action
`commit`	normal end of action (EOA)
`abort`	abnormal end of action

The interactions with the Clouds action manager necessary when such events take place are done by default procedures supplied by the Aeolus compiler and runtime system; these procedures are called *action event handlers*. When an action event occurs for a particular action, the action manager(s) involved invoke the event handlers for each object touched by that action.

As was described above, by use of the `autorecoverable` class of object, the programmer may take advantage of the recovery facilities of the Clouds system by having the compiler generate the necessary code automatically. This automatic recovery mechanism requires recovery of the entire state of the object, and uses the default action event handlers. However, it is sometimes possible for the programmer to improve the performance of object recovery by providing one or more object-specific event handlers which make use of the programmer's knowledge of the object's semantics; these programmer-supplied event handlers then replace the respective default event handlers for that object. Thus, if object class keyword `recoverable` is specified in the definition header of the object being implemented, the programmer may give an optional *action event part* in the object's implementation part. Following the keywords `action events`, the programmer lists the name of each action event handler provided by the object implementation as well as the name of the action event whose default handler the specified handler is to override. Thus, for example, the specification (in an object implementing a bounded-stack abstraction):

```
action events
   stack_BOA overrides BOA,
   stack_nested_precommit overrides nested_precommit
```

indicates that the default handlers for the `BOA` and `nested_precommit` action events are to be replaced by the procedures named `stack_BOA` and `stack_nested_precommit`, respectively, for the bounded-stack object type only.

As mentioned above, if an object being implemented is of class `recoverable`, then some of its variables may be declared in a `recoverable` area. When a nested action first invokes an operation on a recoverable object (''touches'' that object), the action is given a new *version* of the recoverable area which initially has the same value as the version belonging to the action's immediate ancestor. The set of versions belonging to uncommitted actions which have touched a recoverable object is maintained on a *version stack* by a Clouds action manager. When a nested action commits, its version replaces that of its immediate ancestor. When a toplevel action commits, its version is saved to permanent storage. If an action is aborted, its version is popped from the version stack. Thus, recoverable areas (in conjunction with appropriate use of synchronization) provide *view atomicity*, that is, an action does not see the intermediate (uncommitted) results of other actions. Also, the use of recoverable areas allows the programmer to provide finer granularity in the specification of that part of the object state which must be recoverable, since the use of automatic recovery on an object (the `autorecoverable` object class) requires recovery on the entire state of the object. The interaction with the action manager necessary to manage the states of recoverable areas is implemented by the action event handlers as described above. Again, the default event handlers may be overridden by programmer-supplied event handlers for the entire object to achieve better performance.

It may sometimes be desirable to make large data structures resilient. In such cases, the recoverable area mechanism may be inefficient, since it requires the creation of a new version of the entire recoverable area for each action which modifies the area. Often in such cases the programmer may take advantage of knowledge of the semantics of the data structure to efficiently program the recovery of the data structure. The Aeolus language provides two constructs which aid in the custom programming of data recovery, the so-called *permanent* and *per-action variables*, constructs proposed by McKendry [McKe85a].

Any type may be given the attribute `permanent`. This attribute indicates that members of that type are to be allocated on the *permanent heap*, a dynamic storage area in the object storage of each object instance. This area receives special treatment by the Clouds storage manager; in particular, it is *shadow-paged* during the `toplevel_precommit` action event.

Aeolus also provides the per-action variable construct. A per-action variable specification resembles a recoverable area specification, and its semantics is also similar, in that each action which touches an object with per-action variables gets its own version of the variables; however, the programmer may access the per-action variables not only of the current action, but also of the parent of the current action. Also, per-action variables are allocated in *non-permanent storage*, that is, in storage the contents of which may be lost upon node failure. The variables in a per-action specification are accessed as if they were fields in a record described by the specification; two entities of this "record type" are implicitly declared: `Self` and `Parent`, which refer respectively to the per-action variables of the current action and its immediate ancestor.

Permanent and per-action variables may be used together to simulate the effect of recoverable areas at a much lower cost in space per action. In general, the per-action variables are used to propagate changes to the resilient data structure up the action tree; these changes are then applied during the `toplevel_precommit` action event to the actual data structure in permanent storage. The use of permanent and per-action variables is shown more fully in the Aeolus papers cited above.

The right-hand side of an assignment statement may take the form of an *action invocation*. Here, the right-hand side (which consists of an operation invocation which, if the operation is value-returning, is embedded in another assignment statement) is invoked as an action; the *action ID* of this action is assigned to the variable designated by the left-hand side of the action invocation. Thus, for example, if the bounded-stack object mentioned above were defined as a recoverable object, one might invoke one of its operations as an action:

```
aID := action( stack_instance @ push( elem ) )
```

The action ID may be used as a parameter in operations on the action manager which provide information about the status of the action, cause a process to wait on the completion of an action, or explicitly cause an action to commit or abort. By use of additional syntax not shown here, the programmer may specify that an action be created as a "top-level" action, that is, as an action with no ancestors; a top-level action cannot be affected by an abort of any other action. Otherwise, the action is created as a "nested" action, that is, as a child (in the so-called *action tree*) of the action which created it; as described below, a nested action may be affected by an abort of one of its ancestors. Optionally, a *timeout value* may be specified in the action invocation clause; if the action has not committed by the expiration of this timeout, the action will be aborted. If no timeout value is specified, a system-defined default value is used. The detailed semantics of action invocations, and requirements on objects that may have operations invoked as actions, are described in the papers on Aeolus cited above.

3. Overview of Distributed Locking

In this section, we outline a model of concurrency control and replication management for the Clouds system, called Distributed Locking (DL). The linguistic and runtime mechanisms required to support DL are described in the following sections.

In the DL methodology, derivation of a replicated object from its single-site implementation consists essentially of two steps:

1. The user writes a single-site definition and implementation of the object. This implementation includes specification of all lock types used by the object to ensure view atomicity in the presence of concurrently-executing actions.
2. The user writes an *availability specification* (`availspec`) for the object. This specifies the number of replicas of each instance of the object to be generated, the replication control policies to be used, and (optionally) the relative availabilities of the modes of each lock type specified by the object. If no `availspec` is provided, the object is assumed to be nonreplicated.

The `availspec` construct is discussed in detail in Section 4. Note that availabilities are expressed in terms of the modes of locks rather than in terms of operations. Together with the *domain* notion, with which lock granularities are expressed in Aeolus/Clouds, this gives the user more latitude in the expression of relative availabilities than is provided in related work (described in Section 6).

The automation of replication provided by the DL methodology is based on a concept similar to that of *action events* and *object events* as discussed in Section 2. The programmer may specify the interaction of an object with the action management system at critical points in the processing of an action via writing handlers for the action events; handlers for object events allow the object to participate in its creation and destruction. In a similar spirit, we have identified two critical points in the handling of an operation invocation on a replicated object: the *lock event*, during which the invocation attempts to synchronize some subset of the replicas of the object; and the *copy event*, during which the state resulting from the invocation is transmitted to the subset of replicas synchronized during the corresponding lock event. These events correspond to the concurrency control and consistency maintenance aspects of replication control, respectively. Note that the names we have chosen for these events reflect the lock-based synchronization and stable storage-based recovery mechanisms of Clouds; extensions to other synchronization and recovery methods are considered briefly in Section 7. For reasons examined in Section 5, we require that an invocation on a replicated object be made in the context of an action.

Policies for control of concurrency among replicas, and for control of the copying of state among replicas, are expressed in a `lock` object event handler and a `copy` action event handler, respectively, in the `availspec` for an object. Preprogrammed default handlers for these events, implementing commonly-used schemes such as quorum consensus, may be requested by the user if appropriate. If the user wishes to provide application-specific handlers for these events, the same system-provided primitives used in the construction of the default handlers are available for use in programming user-specified handlers. These primitives are described in Section 5, and example event handlers using the primitives are also presented there.

4. Availability Specifications

As discussed in Section 6, the Consensus Locking model of Herlihy allows the specification of the availability properties of an abstract data type in terms of the initial and final quorums required for an operation. It has already been mentioned that in the Distributed Locking model it makes sense to speak of the availability properties of lock modes (rather than of operations, as in other schemes). Some means is needed of allowing the programmer to specify these availability properties for an object without requiring modification of the single-copy version of the object definition or implementation.

In Distributed Locking as implemented in the Aeolus/Clouds system, the availability properties of a replicated object are specified in a separate compiland for that object type, called the *availability specification part* (or `availspec`, for short). The properties specified in an availspec include the number of replicas, the replication management algorithm desired (*e.g.*, quorum assignment, available-copies, etc.), the name of each lock type declared by the implementation of that object along with the names of that lock's modes, and (optionally) the availability relationships among the modes of each lock type used by the implementation of that object. All internal and/or non-Clouds objects used by a replicated object must also have a replication specification; this requirement is applied recursively to these objects. The availability information of a non-Clouds object is inherited by the object which imports it; thus, the effect is as if locks declared by non-Clouds objects were instead declared by the importing Clouds object.

If a voting method is chosen, the quorum assignments for each lock may be derived from the replication specification using integer programming methods. The availability relationships among locking modes, expressed as relative availabilities, may be transformed into constraints on the space of feasible solutions; the objective function may be chosen to maximize the minimum availability over the locking modes subject to these constraints. The construction of this linear program is discussed in more detail later in this chapter. This information is transformed by the Aeolus compiler into a table of replication management information which is stored in the TypeTemplate of the Clouds object (the TypeTemplate is used by the Clouds system to generate instances of an object type). This information is placed in the header information of each object instance and is used by the Distributed Locking primitives to guide the selection of sets of replicas for Distributed Locking (see Section 5).

The Aeolus *availability specification* bears some resemblance to the *fault-tolerance specification* of the HOPS system (cf. Section 6). However, in HOPS the programmer must select among several predefined policies for replication control; there is no provision for user programming of these policies. The ability of the programmer to specify lock and copy event handlers as well as the provision of primitives in support of programming these handlers allows the use of a wider range of replication control policies with the Aeolus availspec construct.

4.1 Example of an Availability Specification

A sample availspec making use of the quorum event handlers is given in Figure 5. This availspec applies to a resilient symbol table object, the definition for which is presented in Appendix A; the implementation of this object is presented and discussed in detail in [Wilk87a]. For the purposes of this example, we will describe only the locks declared in the symbol table implementation.

For synchronization purposes, a lock is declared which allows the entire symbol table to be locked:

```
symtable_lock : lock ( exact : [ exact ], nonexact : [ nonexact ] )
```

Note that operations acquiring symtable_lock in exact mode may run concurrently with other operations acquiring it in exact mode, and similarly for nonexact mode; however, operations attempting to acquire the lock in exact mode must block on those holding it in nonexact mode, and *vice versa*. The use of symtable_lock is to lock out changes during the exact_list operation. Thus, the insert and delete operations acquire this lock in nonexact mode, while the exact_list operation acquires it in exact mode; the lookup and quick_list operations need not acquire the lock at all.

The resilient symtab object must operate in an action environment; thus, additional synchronization is needed to assure the view atomicity of modifications. For this purpose, a new lock variable is introduced which controls the visibility of names in the symbol table:

```
availspec of object symtab ( d : unsigned ) is

    ! Availability specification of the symbol table object using
    ! the quorum consensus scheme.  The DistLock pseudo object
    ! definitions are imported automatically by all availspecs,
    ! but we must import the quorum definitions to use its
    ! predefined handlers.

    import quorum

    ! First, we specify the degree of replication (the number of
    ! replicas).  Here, the degree is taken from an additional
    ! parameter, d, which is specified during creation of an
    ! instance of this object.

    degree is d

    ! The resilient symtab object defines two locks, each with two
    ! modes.  We define the relative availabilities for the modes
    ! of each lock as follows.  The relative availabilities are
    ! used in the constraints of an integer program which is used
    ! in turn to generate the quorum assignments for each lock
    ! mode.

    lock symtable_lock with exact = nonexact

    lock name_lock with read > write

    ! The definitions of the lock and copy events.  Here, we just
    ! use the predefined handlers for quorum consensus.

    availspec events
       quorum_lock overrides lock_event,
       quorum_copy overrides copy_event

end availspec. ! symtab
```

Figure 5. Availability Specification for the Resilient Symbol Table

```
name_lock : lock ( write : []         ,
                   read  : [ read ] ) domain is name_type
```

This lock defines the usual multiple reader/single writer protocol over values of `name_type` (that is, the type of keys). The `insert` and `delete` operations acquire this lock in `write` mode; the `find` operation acquires it in `read` mode. Thus, attempts to insert or delete a given name may not execute concurrently with each other or with attempts to read that name.

The degree of replication (*i.e.*, the number of replicas for a given instance of `symtab`) is given as a formal parameter to the `availspec`; the actual parameter is supplied (in addition to any object parameters specified by the definition part of the object) during creation of object instances.

The `availspec` also specifies the relative availabilities of the modes of each lock declared by `symtab`. Here, the two modes of `symtable_lock` are declared to have the same availability level; however, the `read` mode of `name_lock` is declared to be more available than the `write` mode. The relative availability declarations are used to determine the size of quorums for each mode.

Finally, the alternate handlers for the `lock` and `copy` events are specified. Here, the `quorum_lock` and `quorum_copy` operations made accessible by importing the `quorum` pseudo-object are used.

4.2 Computing Quorum Assignments

When a voting method is used for replication control, the system requires information about the minimum number of replicas required to constitute a quorum for each lock mode. As shown in the example `availspec` in the previous section, the programmer may specify the relative availabilities of the modes of each lock. This information is used to generate constraints for an integer program which computes the actual quorum requirements; the requirements for the modes of each lock of the object are then stored in the object state in an array associated with that lock. A primitive is provided for use in a `lock` event handler which returns the minimum quorum size associated with the lock and mode active at the invocation of the handler (that is, the request for which caused the `lock` event). The Distributed Locking primitives are described in Section 5.

The integer program used to generate the quorum information for each lock is built as follows. If the *i*th variable of the integer program represents the minimum number of replicas required to constitute a quorum for mode *i* of the lock, then the objective function is chosen to minimize the maximum value over all of the variables. As the availability of a mode is inversely proportional to the size of the quorum required for that mode, the objective function has the effect of maximizing the minimum availability over the modes. The relative availabilities of the locking modes as specified by the programmer in the `availspec` are used as constraints on the integer program; if no relative availabilities were specified, the availabilities of the modes are taken to be equal. There are additional constraints generated by the requirement of voting methods that the quorums of each pair of modes intersect (that is, that the sum of each pair of variables be greater than or equal to the degree of replication plus one), as well as that the value of each variable be nonnegative and be less than or equal to the degree of replication.

5. Support for Distributed Locking

As defined in Section 3, the term Distributed Locking refers to a *methodology* for deriving a replicated implementation from its single-copy version, as well as to a *mechanism* to support this methodology. A powerful feature of Distributed Locking is that it does not assume any particular *policy* for replication control. Although the user may easily specify use of one of several default policies in the areas of replica concurrency control and state copying, it also allows the user to explicitly program policies for these purposes. The mechanisms provided by Distributed Locking for support of both default and user-programmed policies are described below.

5.1 Naming Replicated Objects

The mechanism required for support of Distributed Locking requires modifications to the Clouds object naming scheme to support replication.

We have considered two different capability-based naming schemes which may be used in support of *state cloning*, as described in Section 1. The first scheme requires minimal changes to the Clouds kernel, but relies on facets of the Clouds object lookup mechanism which may not be applicable to other systems. In Clouds, the search for an object begins locally (that is, on the node which invoked the search), and—if the object is not found locally—proceeds to a broadcast search. If the internal objects belonging to a replica are constrained to reside on the same node as their parent object, then the local search will locate the local instance of the internal object. (This constraint is not considered to be onerous, since the internal objects of each replica need to be highly available to that replica in any case, and thus should logically reside on the same node as the parent replica.) Thus, each replica of an object (each of which resides on a separate node) may maintain its set of internal objects using the same capabilities as each other replica. (This situation may be created by initializing one replica, and then cloning its state to the other replicas.) Although there will thus be multiple instances (on separate nodes) of internal objects referenced by the same capability, there should be no problems caused by this, since—by the definition of internal object—only the parent object or its internal objects may possess the capability to an internal object, and the object search will always locate the correct (local) instance. Thus, state cloning may be used to copy the state of a replica to the other replicas without causing the problems with respect to internal objects described in Section 1 (concerning references to internal objects contained in the replica's state), since under this scheme all replicas may use the same capabilities for referencing internal objects. This scheme is an extension of a facility already supported by the Clouds kernel for cloning read-only objects such as code. This scheme is called *vertical replication*, since it maintains the grouping of internal objects with their parent object.

The other naming scheme makes fewer assumptions about the lookup mechanism than vertical replication, but requires more kernel modifications. In the second scheme, each instance of the replicas' internal objects is again named by the same capability, at least as far as the user is concerned; however, the kernel maintains several additional bits associated with each capability identifying a unique instance. (These additional bits may be derived, for example, from the birth node of the instance.) When a (parent) replica invokes an operation on an internal object, the kernel selects one of the replicas of the internal object according to some scheme (*e.g.*, iteration through the list of nodes containing such objects until an available copy is located). Thus, a set of replicas of internal objects is maintained in a "pool" for access by all parent replicas. Again, each parent appears to use the same (user) capability to reference a given internal object, so the problems of state cloning disappear. Since this scheme maintains a logical grouping of the copies of an internal object, rather than grouping internal objects with their parent object, this scheme is called *horizontal replication*. One such naming scheme is described in a paper by Ahamad *et al.* [Aham87a]

The attractions of the vertical replication scheme are that it is conceptually simple, that it requires no modifications to the kernel capability-handling mechanisms, and that, by requiring coresidence, it enforces a property which enhances availability. To see this, recall that independent failure modes are desirable among different replicas of a replicated object, since the probability that the replicated object will be available is the probability that *any one* of the set of replicas will be available. On the other hand, dependent failure modes are desirable among a given replica and its internal objects, since the probability that the given replica will be available is the probability that *all* of the set of internal objects will be available. Requiring coresidence of objects related by logical nesting introduces dependence of their failure modes.

Unfortunately, the vertical replication scheme is not viable in general, since the coresidence requirement may sometimes be unrealistic. It may sometimes be the case that it is impossible to satisfy coresidence, due to the size of nested objects (making it impossible to accommodate them on the same node), or due to insufficient space because of previously-existing objects on that node. Thus, vertical replication must be abandoned as lacking sufficient generality in its applicability. Fortunately, the horizontal replication scheme does not share this drawback.

The horizontal replication scheme has been further developed in a recent paper by other researchers on the Clouds project [Aham87b]. However, the invocation scheme may be altered to take advantage of coresidence when possible. The search scheme used for invocation of replicated objects in the paper cited above involves a random choice among the set of replicas. This differs markedly from the current Clouds search scheme for non-replicated objects, which is essentially as follows:

```
if <object found locally> then
   <perform invocation on local object>
else
   <perform global search>
end if
```

This search scheme may be modified to take advantage of coresidence as follows:

```
if <object found locally> then
   <perform invocation on local object>
else
   if <object is replicated> then
      <select randomly among the set of replicas>
   else
      <perform global search>
   end if
end if
```

Note that, if only one replica is stored per node, the local search involves only the so-called "user capability;" that is, it does not involve the extra bits used by the "kernel capability" to distinguish among replicas. If one allows more than one replica per node, some use of the kernel capability must be made to select an appropriate instance; this may require specific knowledge of which replicas are stored at which nodes.

5.2 Invocation of *Lock* and *Copy* Events

Support of the Distributed Locking mechanism requires modification of the Aeolus/Clouds object and action management facilities in two areas.

1. When an operation attempts to obtain a lock on an instance of a replicated object, locks are obtained at some appropriate subset of its replicas, by invoking the *lock* event handler on that object. (Using terminology introduced by Ahamad and Dasgupta [Aham87b], the replica at which the original invocation took place is called the *primary cohort* [*p-cohort*]; the other members of the locked subset of replicas are called *secondary cohorts* [*s-cohorts*].)
2. During the handling of the precommit event of the controlling action, the state of each p-cohort touched by that action is copied to its s-cohorts, by invoking the *copy* event handler on each p-cohort.

In Section 1, two methods of copying object state applicable to the Clouds model were identified:

1. *idemexecution*, or execution of an invocation at each member of the set of replicas; and

2. *cloning*, or execution of an invocation at a single replica, and then explicitly copying its state to the other replicas.

Because of the drawbacks of idemexecution (including the possibility of repeated invocations on objects external to the replicated object, as well as the difficulty of handling invocations with non-deterministic results in this scheme), the most viable mechanism seems to be cloning. However, the Distributed Locking mechanism does not preclude the use of idemexecution in the *copy* event, and provides primitives for its support.

Since a replicated object may have an arbitrary structure of logically nested objects, it is a non-trivial problem to determine exactly what state of which objects must be copied to implement a cloning operation. That is, it does not suffice to merely copy the state of the *p-cohort* to its *s-cohorts*; the states of all objects nested with respect to the *p-cohort* which were involved in the given operation must also be copied to their respective replicas (the nested objects of the *s-cohorts*). Fortunately, the Clouds action mechanism provides a means of determining which objects must be cloned: the action manager maintains a list of objects *touched* by an action. (This is the reason behind requiring that invocations on replicated objects take place in the context of an action.) Indeed, one need only perform cloning upon commit of an action, since the results of an action become visible to other actions only after commit. At that time, the so-called "shadow set" of each touched object is available. (In very simplified terms, this is the set of pages in the object's recoverable area which have been modified by the action.) If the constraint is made that all replicated objects be recoverable, then to implement cloning, one need only copy the shadow set of each touched object to the other replicas in that object's set, and perform the commit actions of storage management at each replica. The shadows are committed at each of the s-cohorts as if the shadows had been produced by execution at that s-cohort.

5.3 Primitives for *Lock* and *Copy* Event Handlers

If the user wishes to provide application-specific handlers for these events, the same system-provided primitives used in the construction of the default handlers are available for use in programming user-specified handlers. These primitives, and their purposes, include those for such purposes as:

- acquisition at a specific replica of the currently-requested lock (with the same mode and value, if any), for implementing lock propagation;
- invocation at a specific replica of the same operation (with the same parameters) requested at the current replica, for implementing idemexecution;
- broadcast of state shadow sets to all replicas holding a specified lock (with a specified mode and value), for implementing cloning via shadows; and
- invocation at a specific replica of an arbitrary operation, for implementing cloning via logs or state reconciliation strategies.

The intention is to provide facilities at a level sufficiently low to accommodate all schemes of interest. Some other useful predefined objects, such as those implementing list abstractions, are available for such purposes as maintaining and traversing the list of replicas at which locks have been obtained (and to which the object state must later be copied).

The primitives described above are encapsulated in an Aeolus pseudo-object called *DistLock*. The definition of *DistLock* is presented in its entirety in Appendix B.

```
implementation of pseudo object quorum is

   ! Here, we define handlers for the lock and copy events which
   ! implement quorum consensus.  This pseudo object is imported
   ! by any availspec wishing to use its predefined handlers.

import DistLock

procedure quorum_lock () is
   ! A simple-minded lock event handler for quorum consensus.
   ! Locks are obtained on at least a minimum quorum assignment
   ! specified by the assignment matrix generated by the
   ! importing availspec.

   this_version ,
   max_version  : version_number
   num_locked   ,
   good_replica : replica_number

   begin
      ! Find out how many replicas have been locked already by
      ! the current action.
      num_locked := DistLock @ currently_locked()

      ! Initially, the latest version seen is set to this
      ! instance's version number.
      max_version  := DistLock @ my_version()

      ! Attempt to lock all available replicas.
      for r in replica_number[ 1 .. DistLock @ degree() ] loop
         if DistLock @ lock_replica( r, this_version ) then
            num_locked += 1
            if this_version > max_version then
               max_version  := this_version
               good_replica := r
                  ! remember which replica has the latest version
            end if
         end if
      end loop

      ! At least a quorum of replicas must have been locked.  If
      ! not, abort the invoking action.
      if num_locked < DistLock @ quorum_size() then
         Abort_Myself()
      end if

      ! If there is a later version of the state than that of
      ! this replica, copy it here.  (This updates the local
      ! version number.)
      if good_replica <> DistLock @ my_replica() then
         if not DistLock @ get_state( good_replica ) then
            Abort_Myself()  ! replica was unavailable
         end if
      end if

      ! Copy the local state to all replicas which have version
      ! number less than that of the local copy.
      for r in replica_number[ 1 .. DistLock @ degree() ] loop
         if not DistLock @ send_state( r ) then
            Abort_Myself()  ! replica was unavailable
         end if
      end loop
   end procedure ! quorum_lock


procedure quorum_copy is
   ! The copy event handler for quorum consensus.  The shadow set
   ! is copied to the set of replicas locked in the lock event.

   begin
      if not DistLock @ broadcast_shadows() then
         Abort_Myself()  ! copy was unsuccessful
      end if
   end procedure ! quorum_copy

end implementation. ! quorum
```

Figure 6. Lock and Copy Event Handlers for Quorum Consensus

5.4 Examples of Event Handlers in Distributed Locking

A sample implementation of `lock` and `copy` event handlers using the General Quorum Consensus algorithm are given in Figure 6. The treatment of these event handlers has been kept on a fairly naive level to avoid obscuring neither the general lines of the algorithm used nor the use of the Distributed Locking primitives. The handlers are encapsulated in a pseudo-object called `quorum` which may be imported by an `availspec` in order to use its handlers.

As described in a previous section, the replica of an object at which an operation is invoked is called the *primary cohort* or *p-cohort*; a request for a lock at the p-cohort causes its lock event handler to be activated. The handler for the `lock` event, here called `quorum_lock`, attempts to lock each other available replica (called *secondary cohort* or *s-cohort*) by use of the `lock_replica` Distributed Locking primitive; if successful, this primitive returns the version number of the new s-cohort as an `out` parameter. The maximum version number over all s-cohorts is determined and compared with the version number of the p-cohort; if the latter is not the latest version, the state of the s-cohort having the latest version is copied to the p-cohort. In any case, at this point the latest state is copied to all s-cohorts having earlier states. If the number of s-cohorts is not at least as great as the quorum assignment for the requested lock mode, the enclosing action is aborted.

When the action enclosing the operation invocation prepares to commit, the `copy` event handler (here called `quorum_copy`) is activated. This handler uses the `broadcast_shadows` primitive to copy the shadow set (of changed pages) of the p-cohort to the s-cohorts locked in all activations of the `lock` event handler by the current action. If the copy is successful, the shadow sets are committed at the s-cohorts as well as the p-cohort to yield the updated state.

There are obvious improvements which might be made to this simple version of `quorum`. For example, `quorum_lock` relies on the `lock_replica` primitive to "fall through" when an attempt is made to lock a replica which is already an s-cohort. A more sophisticated implementation could maintain a set of replica numbers representing the current set of s-cohorts in order to avoid the overhead of a remote invocation for each redundant `lock_replica` call.

The use of the `broadcast_shadows` primitive in `quorum_copy` requires that the states of all s-cohorts be identical to that of the p-cohort when the `lock` event handling is complete, so that the shadow set broadcast during the `copy` event can be committed into a common permanent state at each replica; this is achieved by copying the state of the replica with the latest version number to those replicas with earlier versions of the state. This implementation assumes that it is uncommon for the version number of a replica to be "out of synch" with its fellow replicas, which is a reasonable assumption if most, if not all, replicas are available to become s-cohorts during each `lock` event. If this assumption is invalid, it may be more efficient to avoid copying of the latest state to the s-cohorts during the `lock` event *and* copying shadow sets during the `copy` event by copying the entire state of the p-cohort to the s-cohorts during the `copy` event.

6. Related Work

In this section, previous work on the properties of resilience and availability in distributed applications is examined. The issues of resilience in work related to Clouds have been examined in previous Clouds dissertations [Allc83b, Spaf86a, Pitt86a]; the discussion in this chapter thus concentrates on the issues of availability as treated by other researchers, except where the previous work relates to the linguistic support for resilience as provided in Aeolus.

The study of the use of replication to enhance availability first occurred in the area of distributed database systems, and was later adopted in the area of distributed operating systems. Thus, the problems in the control of concurrency among replicated objects were studied and, for the most part, solved by database researchers; the concurrency control methods used by the operating systems projects described below are largely derived from the database research. A survey of this work appears in a recent book by Bernstein *et al.* [Bern87a] The history of these efforts is also summarized by Wright [Wrig84a]. However, the research in database systems has been limited to consideration of "flat" objects, such as records or files; as was shown in Section 1, the generalization to arbitrary structure of objects in distributed operating systems research leads to problems related to the mechanisms used for the copying of state among replicas.

6.1 Replication in Database Systems

As with most of the topics involved in the study of distributed systems, the synchronization and recovery of replicated data was first studied in the area of distributed database systems. Examples of database concurrency control methods methods are voting schemes [Giff79a, Thom79a], available-copy methods [Good83a], primary copy methods [Ston79a], and token-passing schemes [LeLa78a]. The intent of these methods is to ensure consistency of the replicated data by requiring access to a special copy or set of copies of the data during failures or partitions. Primary copy methods allow access to a copy during a network partition only if the partition possesses the designated primary copy of the data. Token-passing schemes are an extension of primary copy methods; a token is passed among sites holding a copy of data, and that copy at the site currently holding the token is considered the primary copy. Yet another extension of primary copy methods are the voting schemes. Each copy of the data object is assigned a (possibly different) number of votes; a partition possessing a majority of the votes for that object may access it.

Finally, available-copy methods follow a "read-one, write-all-available" discipline. A **read** operation may access any *initialized* copy (that is, one which has already processed a **write** operation). A **write** operation must access all copies; those which are unavailable for writing are called *missing writes*. A validation protocol, which runs after all **reads** and **writes** of a transaction have either been processed or timed out, guarantees one-copy serializability. This protocol ensures that all copies for which missing writes were recorded are still unavailable, and that all copies accessed are still available. Several researchers have recently proposed enhancements to the original available-copies algorithm [Skee85a, El-A85a, Long87a].

El Abbadi has recently proposed a paradigm for developing and analyzing concurrency control protocols for replicated databases, especially those handling partition failures [El-A87a]. He has also proposed a new protocol, developed within this paradigm, which allows read and write access to data despite partitions.

6.2 Replication in Operating Systems

Previous work in the area of replication of data in distributed operating systems includes the ISIS system at Cornell, the Eden system and the Emerald language at the University of Washington, the Argus system at MIT, Cooper's work on the Circus replicated procedure call facility at Berkeley, the HOPS project at Honeywell, Inc. and Herlihy's work at MIT (General Quorum Consensus) and CMU (Avalon).

6.2.1 ISIS The ISIS system developed at Cornell [Birm84a, Birm85a] supports *k-resilient* objects (objects replicated at $k+1$ sites and which can tolerate up to k failures) by means of checkpoints and the "available copies" algorithm. ISIS objects can refer to other objects, although apparently all such "nested" objects are considered to be external. This system provides both availability and *forward progress;* that is, even after up to k site failures, enough information is available (at the remaining sites possessing an object replica) that work started at the failed sites can continue at these remaining sites. This is accomplished through a *coordinator-cohort* scheme, where one replica acts as master during a transaction to coordinate updates at the other, "slave" replicas ("cohorts"). The choice of which replica acts as coordinator may differ from transaction to transaction. The object state is apparently copied from the coordinator to the cohorts via a cloning operation; this operation has been described as propagating a checkpoint of the entire coordinator [Birm84a], or, in a more recent paper, as propagating the most recent version in a version stack [Birm85a]. In the current system, it is assumed that the network is not subject to partitioning.

In ISIS, a transaction is not aborted when a machine on which its coordinator is running fails (transactions are usually aborted only when a deadlock situation arises). Rather, the transaction is resumed at a cohort from the latest checkpoint, in what is called *restart mode;* this cohort becomes the new coordinator. Operations which the coordinator had executed after the latest checkpoint took place must be re-executed at the new coordinator.

In the course of an operation on a k-resilient object, the coordinator may perform operations on other objects to which it contains references. Such operations on "nested" objects are called *external actions*. Inconsistencies can arise due to external actions performed during restart mode; operations performed on external objects by the new coordinator in this mode were also performed by the old coordinator before it failed. Thus, unless the operations on external objects are idempotent, inconsistencies can arise. (This problem is closely related to the problem of idemexecution on external objects, discussed in Section 1.) This problem is solved in ISIS by requiring external objects to retain results of operations; these retained results are associated with a transaction ID. When a new coordinator takes over from a failed coordinator and enters restart mode, it uses the same IDs for its external operations, and rather than re-execute these operations, the external objects merely return the associated results.

There is also an idemexecution scheme due to Joseph [Jose85a, Jose86a] which was apparently implemented as an experiment using the ISIS system as a testbed, rather than as part of the ISIS replication mechanism itself. In Joseph's scheme, the coordinator performs the requested operation, and then instructs its cohorts to perform the same operation.

Recently, a new version of the ISIS system, called ISIS-2, has been designed; it is anticipated that this new system will be operational by Fall 1987. The ISIS-2 design exploits a new abstraction called the *virtually synchronous process group* [Birm87a]. In this abstraction, a distributed set of processes cooperate to perform work in an environment in which broadcasts, failures, and recoveries are made to appear synchronous.

6.2.2 Eden and Emerald The Eden system [Alme83a, Blac86a] was under development at the University of Washington from September 1980 until late 1986; the system has been operational on a collection of VAX systems (and later Sun workstations) since April 1983. Support in the Eden system for replication has been studied at both the kernel level and the object level. The kernel level implementation of replication support is called the *Replect* approach (for replicated "Ejects," or Eden objects), while the object level implementation is called *R2D2* (for "Replicated Resource Distributed Database"). Both implementations use quorum consensus for concurrency control.

In Eden, objects are *active*, that is, each object encapsulates—besides data and operations on the data—one or more active processes which are permanently associated with that object. Normally, an object has two forms: an *active form* (AF) which exists in volatile memory, and a *passive representation* (PR), which is a checkpoint of the AF on disk. The PRs are maintained in *permanent object databases* (PODs), one of which exists on each node in the system. In the unreplicated case, an object has only one AF and one PR at any time.

In the Replect approach [Prou85a], although the PR of the object is replicated, the object still has only one AF at any time. Thus, one capability is used to refer to all replicas of the object. Hence, a Replect is referenced by the user in the same way as a normal object; the Replect mechanism is transparent to clients of a Replect. A transaction management facility is required to ensure that multiple AFs are not produced by competing transactions despite crashes. (The basic Eden system does not provide a transaction management facility.) Updates are performed by selecting one of the PODs to act as transaction manager in a master/slave protocol.

In the R2D2 approach [Noe85a], each replica is a complete object, consisting of an AF as well as a PR. Each replica is unaware of the others, but clients must refer to the replicated object by using a set of capabilities (to the multiple AFs), one for each replica; thus, this mechanism is less transparent than the Replect approach. R2D2 objects are stored in a replicated hierarchical directory structure. Invocations on objects replicated using R2D2 must use a specialized transaction manager (called R2D2TM), which traverses the replicated directory and handles the multiple updates on members of the set of replicas. Members of the set which are unavailable due to crashes are replaced via *regeneration*. The level of the directory in which the unavailable object is maintained must be updated to reflect the replacement.

The basic Eden system was not designed to handle partitions [Noe85a]. The two replication approaches described above compensate for this lack in differing degrees. Using the R2D2 approach, an object will be able to regenerate if its partition contains a copy of the PR and a suitable number of machines, and will then be able to continue to operate. However, upon the resolution of the partition, the states of competing versions of the object must be merged. Thus, the Eden authors prefer to use voting methods, allowing simple merging of partitions, although replicas in a partition without a majority will be unable to operate. Using the Replcct approach, on the other hand, problems arise even with voting methods due to the problem of avoiding having multiple AFs. If a partition contains a quorum of PRs, but the AF is gone, it is not possible to tell if the AF is inactive (dead) or in another partition. If one is to allow multiple AFs, a state-merging scheme must also be provided, since the isolated AF may be updated with no attempt to checkpoint to the PRs.

No mention is made in the Eden references of support for arbitrary structure of objects or of the associated problems of state propagation.

Another project at the University of Washington is concerned with the design and implementation of an object-oriented language for distributed applications [Blac86b, Blac87a]. Emerald provides a hierarchy of object classifications similar to that provided by Aeolus (as described in Section 2); however, selection of an appropriate classification for an object is made automatically by the Emerald system. Emerald does not at present provide support for fault tolerance.

6.2.3 Argus The Argus system at MIT [Lisk87a, Lisk84a, Lisk83a, Weih83a] is a language and system for distributed applications which has evolved from the CLU language. Argus provides an object construct (called *Guardian*) which encapsulates data and processes, giving an abstraction of a physical node or server. Argus also retains the *cluster* construct from CLU, which provides functionality similar to that of local objects in Aeolus; however, the syntax of Guardians is not similar to that of clusters. Resilience in Argus is based on the notion of system-provided primitive *atomic data types*, from which user-defined atomic data types may be constructed. These primitive atomic data types also define the synchronization properties of the user-constructed types. Experience with programming a distributed, collaborative editing system in Argus has been described by Greif *et al.* [Grei86a]; one criticism arising out of this experience was that they were sometimes forced to use a Guardian where a cluster might have been more appropriate.

Recent work at MIT has been concerned with availability issues in distributed services [Lisk86a, Lisk87b]. The researchers have developed a method for constructing highly-available services which maintain a form of view atomicity despite the presence of old information in their states. This method requires that the properties of the information be *stable* in the sense that once a property becomes true, it does not change thereafter. Availability methods possessing this property are useful in applications such as distributed garbage collection.

6.2.4 Circus Cooper has investigated a mechanism called the *replicated procedure call,* which he implemented at Berkeley in a system called *Circus* [Coop84a, Coop85a]. In Cooper's scheme, although replicas of an object have no knowledge of each other, they are bound (via run-time support) into a server called a *troupe* which may be accessed by client objects. (The client objects know that the server is replicated.) An object in Circus may have arbitrary structure, containing references to both internal and external objects. However, the object is currently required to be deterministic. His scheme uses idemexecution for state propagation. When a troupe accesses an external troupe (a so-called "many-to-many" call), results of operations on objects of the server troupe are retained by the callees; these results are associated with *call sequence numbers,* and are returned when subsequent calls by the replicas of the caller troupe with the same sequence numbers are encountered, thus avoiding the inconsistencies possible with idemexecution on external objects. Concurrency control is by majority voting. Thus, if a partition does not have a majority of troupe members, invocations will not be able to proceed.

6.2.5 The HOPS Project The *Honeywell Object Programming System* (HOPS) [Hone86a] under development at Honeywell, Inc., has research goals similar to those of our methodology research. The stated goals of the HOPS project are:

- to alleviate what is seen as a lack of experience in the field of distributed systems in implementing mechanisms which perform failure detection, failure recovery, and resource reconfiguration;
- to provide programming support for developing fault-tolerant distributed applications; and
- to assess the actual benefits and costs of such mechanisms in terms of performance, reliability, and availability.

HOPS consists of an implementation language derived from Modula-2 together with a distributed runtime support system. The language requires that HOPS objects (or *HOPjects*) be specified in three parts: an interface specification, a body (or implementation specification), and a *fault tolerance specification*. In the latter, the programmer may specify attributes and policies relating to recovery, concurrency control, and replication which are to be used for that object, thus giving the programmer a choice among several mechanisms provided by HOPS in each of these areas. The distributed runtime system (together with the underlying host operating system) provides facilities for naming and addressing objects, communication, failure detection and recovery, local and distributed transaction management, concurrency control, recovery, and replication. HOPS is currently being implemented on a network of Sun-3 workstations under the Sun version of Unix 4.2.

Mechanisms for achieving fault tolerance in HOPS include the *distributed recovery block* (DRB) mechanism and *distributed conversations*. (The recovery block and conversation mechanisms are described in detail in a book by Anderson and Lee [Ande81a] as well as in the HOPS report cited above.) Basically, the combination of the DRB and conversation mechanisms provide fault tolerance by what is essentially "software modular redundancy." Processes at two or more nodes execute one of a set of differing sections of code (called *try blocks*) which implement the same specified function; the results of these try blocks must pass the same *acceptance test* (possibly with majority voting), or the participating processes are rolled back to a checkpoint (called a *recovery line*) and retry the computation with their alternate try blocks. Thus, both fail-stop and some Byzantine-style failures may be detected and tolerated by this scheme.

6.2.6 General Quorum Consensus and Avalon Herlihy's work on General Quorum Consensus [Herl84a] concerns the extension of quorum intersection methods to take advantage of the semantic properties of abstract data types. Previously, work on quorum methods—mostly in the database area—has been limited to a simple read/write model of operations. Herlihy's extensions allow the selection of optimal quorums for each operation of an abstract data type based on the semantics of that operation and its interaction with the other operations of the data type.

Herlihy's method is based on the analysis of the algebraic structure of abstract data types. This entails the construction of a "quorum intersection graph," each node of which represents an operation of the data type, and each edge of which is directed from the node representing an operation *O1* to the node representing operation *O2*, where each quorum of *O2* is required to intersect each quorum of *O1*. From the quorum intersection graph, optimal quorums for each operation may be calculated, given the number of replicas of the data, and the desired availability of each operation in relation to the other operations of the data type.

Herlihy shows that his method can enhance the concurrency of operations on replicated data over that obtained from a read/write model of operations. He also claims advantages for his methods in the support of on-the-fly reconfiguration of replicated data, and in enhancing the availability of the data in the presence of network partitions.

More recently, Herlihy has developed two new methods for integrating concurrency control and recovery for abstract data types, called *Consensus Locking* and *Consensus Scheduling*. In these schemes, Herlihy requires that the quorum intersection relation and the lock conflict relation (the complement of the lock compatibility relation) for an object satisfy a common *serial dependency relation* on that object; he notes that, in practice, the lock conflict and quorum dependency relations will be the same [Herl85a]. A detailed comparison of Consensus Locking is presented in [Wilk87a].

A third scheme, called *Layered Consensus Locking*, extends the Consensus Locking method by associating a *level* with each activity in the system [Herl85b]. Activities at a higher level are serialized after activities at a lower level. If an activity executing at a given level is unable to make progress after a failure with its current quorum assignment, it may restart at a higher level and switch to another quorum assignment. Each initial quorum for an invocation at level n is required to intersect with each final quorum for an event at levels $<= n$.

Herlihy and Wing recently have been developing a set of linguistic features, called *Avalon*, for support of transaction processing [Herl87a]. Avalon is intended to be implemented as extensions to pre-existing languages such as Ada and C++, and is built on the *Camelot* distributed system developed at CMU. Avalon provides support for action event handling resembling that provided by Aeolus, as described in Section 2. Avalon also provides support for testing serialization orders dynamically.

7. Conclusions and Future Directions

In this paper, methods of achieving resilience and availability in the Clouds system have been examined. In the course of this work, we have designed a systems programming language providing access to the Clouds features of objects and actions, features which—used in conjunction with the Aeolus runtime support—provide powerful support for resilience of data and computations. Although automatic support for resilient objects—the paradigm provided by other systems with goals similar to those of Clouds—is provided as an option in Aeolus, facilities are also provided that allow the programmer to specify resilience mechanisms more appropriate to the semantics of the object when desirable.

We have taken a similar approach in designing a scheme, Distributed Locking, for supporting high availability of Clouds objects. Most distributed system projects providing support for replication assume a certain policy for replication control, usually quorum consensus. In the course of recent research, several algorithms for replication control displaying availability properties more desirable than those of other algorithms in some situations have been proposed. Thus, it seemed advisable to provide the capability of supporting several different policies for replication control rather than assuming any one policy. Predefined policies may be accessed as defaults if the programmer so desires; however, since a replication control scheme other than one of those foreseen as a pre-programmed policy may prove more appropriate to the semantics of a given object, our scheme also allows the programmer to develop new policies using the same library of support primitives used to develop the default policies.

7.1 Performance of Distributed Locking

We consider the Distributed Locking mechanism in the form described in this paper to be a tool for research into replication techniques rather than a production system for real-world applications. However, it may be instructive to estimate the performance of the mechanism in a sample application in order to demonstrate how such estimates may be derived in other cases;

these derivations would be useful primarily for comparison of different replication techniques. As a sample application, we assume a replicated object of degree three, each replica having a permanent storage area consisting of ten pages, and using the quorum consensus handlers (as described in Section 5) for replication control. For simplicity, we also assume that the action being performed on the replicated object consists of an operation invocation that does not visit other objects, and that this operation causes the entire permanent storage area to be shadowed (the worst case).

The two-phase commit protocol in Kenley's action management design [Kenl86a] requires a total of four message/acknowledgment pairs per site visited by an action. In Phase I (the Prepare phase), the coordinating site must send each visited site a *prepare* message; if all goes well, each visited site responds with a *prepared* message indicating success. In Phase II (the Completion phase), if all visited sites have responded positively to the prepare message, the coordinator sends a *commit* message to each visited site; if commit is successful, each visited site responds with a *committed* message.

In the Clouds prototype, a message/acknowledgement pair for a message of maximum size 1.5 Kbytes requires approximately thirty milliseconds [Stri88a]. (The network driver has not yet been examined for possible performance improvements.) Thus, the messaging overhead of an action commit is 120 ms per site, to which must be added 60 ms for the action manager to write a commit log at the coordinating site.

Timings for writing to stable storage in the Clouds prototype have been measured by Pitts [Pitt86a]. To install the shadow version, there is a constant overhead of approximately 120 ms; there is also a cost per page of the shadow set which ranges between 25 ms (if the write is sequential and does not require a seek) and 51 ms (if the write is random). (These figures are based on a driver for a relatively small, slow disk; a driver for a much faster disk has recently been developed, and should yield much better performance figures, perhaps one-third or better of those of the slow disk.)

If a communications environment is assumed that does not allow broadcast, then messages must be sent separately by action management to each replica of an object touched by an action to perform a commit. If the quorum consensus protocol is used, separate messages must also be sent to each replica to transmit the shadow set of the coordinating site (*p-cohort* in the terminology of Section 5) to the other replicas (*s-cohorts*); a maximum of three 512-byte pages may be transmitted per message. However, stable storage processing may be done concurrently at each replica once the shadow set is transmitted. Let R represent the degree of replication of the invoked object, and P be the number of pages of permanent storage in the object. Then, for a non-broadcast environment, the overhead of the quorum consensus copy event, *i.e.*, the time required to commit the simple action invocation described above on a replicated object, is given by:

$$120R+60 + 30\lceil P/3\rceil(R-1) + 51P+120$$

where the first term represents the contribution by action management overhead, the second term the time required to transmit the shadow set to the s-cohorts (excluding the p-cohort), and the third term the time required to write the shadow set to stable storage at each replica (assuming the worst case in which all writes are random); all constants are in milliseconds. For the sample application described above (where R=3 and P=10), a commit of the simple action would require approximately 1290 ms.

In the Clouds prototype, the action management messages as well as the shadow sets may be broadcast to the replicas, thus eliminating the need for sequential messages to each replica. In this environment, the overhead of the `copy` event reduces to:

$$120+60 + 30\lceil P/3 \rceil + 51P+120$$

and the `copy` event of the sample application would require approximately 870 ms in the worst case. (If the estimates given above for the performance of the faster disk are assumed, the overhead becomes approximately 510 ms.) Note that this expression does not depend R, the number of replicas. The expression is indeed close to the overhead involved in committing a single-site object on a different site than the coordinator for the action; the expression for the single-site case does not include the second term (the overhead of broadcasting the shadow set to the s-cohorts). The time to commit the single-site object is thus approximately 750 ms for the slow disk in the worst case. If all writes on the slow disk were sequential (perhaps a more normal case), the overheads would be 610 ms for the replicated object vs. 490 ms for the single-site object.

A similar analysis may be performed of the additional activity required during the `lock` event handling. Considering the handler for quorum consensus, the worst case occurs when all replicas are available to be locked (requiring R-1 messages to perform the locking), and when the latest version must be copied to all s-cohorts (requiring $\lceil P/3 \rceil(R-1)$ messages). The overhead for the sample application in this worst case would thus be approximately 300 ms. If the state is up-to-date at all cohorts at the time of the `lock` event, however, the overhead would reduce to just that involved for the locking messages, in this case 60 ms.

7.2 Current Status

The first prototype of the Clouds operating system has been implemented and is operational. This version is referred to Clouds v.1. This is being used as an experimental testbed by the implementors. Results of performance tests with this prototype are available in other publications on Clouds [Spaf86a, Pitt86a], and are summarized in [Dasg88a]. The experience with this version has taught us that the approach is viable. It also taught us how to do it better.

The lessons learned from this implementation are being used to redesign the kernel and build a new prototype. The basic system paradigm, the semantics of objects, and the goals of the project remain unchanged and v.2. will be identical to v.1. in this respect.

The structure of Clouds v.2. is different. The operating system will consist of a minimal kernel called *Ra*. Ra will support the basic function of the system, that is location independent object invocation. The operating system will be built on top of the Ra kernel using system level objects to provide systems services (user object management, synchronization, naming, atomicity and so on).

The Ra implementation is now in progress. The action management subsystem, the design of which is described in Kenley's thesis [Kenl86a], is being redesigned to work with Ra. A compiler and runtime system for the Aeolus language have been implemented in a Pascal variant (with some C and assembler in the runtime system); the compiler is being rewritten in Aeolus for portability purposes. Implementation of Distributed Locking will be possible once the redesigned action management subsystem is in place, as the interfaces to action management from Aeolus already exist.

7.3 Future Directions

The version of the Distributed Locking scheme described in this paper is based on the policies of lock-based synchronization and stable storage-based recovery, implemented by the action management and storage management subsystems of Clouds, respectively. As mentioned in Section 2, the Clouds kernel is designed so that these subsystems may be replaced with others implementing different policies. We are currently considering the effects on the DL mechanism of the replacement of locks with timestamp-based synchronization, and the replacement of shadowed stable storage with log-based recovery. We anticipate that these changes will require additions to the library of primitives supporting DL.

In addition, we are considering the effects on DL of relaxing the fail-stop assumption. This will require primitives supporting the reconciliation of replica states which have diverged via operating in separate partitions. These primitives may be used in conjunction with the *reinitialization* object event described in Section 2.

REFERENCES

[Aham87a] Ahamad, M., P. Dasgupta, R. J. LeBlanc, and C. T. Wilkes. "Fault-Tolerant Computing in Object Based Distributed Operating Systems." *PROCEEDINGS OF THE SIXTH SYMPOSIUM ON RELIABILITY IN DISTRIBUTED SOFTWARE AND DATABASE SYSTEMS* (IEEE Computer Society), Williamsburg, VA (March 1987): 115-125.

[Aham87b] Ahamad, M. and P. Dasgupta. "Parallel Execution Threads: An Approach to Fault-Tolerant Actions." TECHNICAL REPORT GIT-ICS-87/16, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, March 1987.

[Allc82a] Allchin, J. E. and M. S. McKendry. "Object-Based Synchronization and Recovery." TECHNICAL REPORT GIT-ICS-82/15, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1982.

[Allc83a] Allchin, J. E. and M. S. McKendry. "Synchronization and Recovery of Actions." *PROCEEDINGS OF THE SECOND SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING* (ACM SIGACT/SIGOPS), Montreal (August 1983).

[Allc83b] Allchin, J. E. "An Architecture for Reliable Decentralized Systems." PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1983. (Also released as technical report GIT-ICS-83/23.)

[Alme83a] Almes, G. T., A. P. Black, E. D. Lazowska, and J. D. Noe. "The Eden System: A Technical Review." TECHNICAL REPORT 83-10-05, University of Washington Department of Computer Science, October 1983.

[Ande81a] Anderson, T. and P. A. Lee. *Fault Tolerance, Principles and Practice.* Englewood Cliffs, NJ: Prentice-Hall International, 1981.

[Bern87a] Bernstein, P. A., V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Reading, MA: Addison-Wesley, 1987.

[Birm84a] Birman, K. P., T. A. Joseph, T. Raeuchle, and A. El-Abbadi. "Implementing Fault-Tolerant Distributed Objects." *PROCEEDINGS OF THE FOURTH SYMPOSIUM ON RELIABILITY IN DISTRIBUTED SOFTWARE AND DATABASE SYSTEMS*, Silver Spring, MD (October 1984): 124-133.

[Birm85a] Birman, K. P. "Replication and Fault-Tolerance in the ISIS System." *PROCEEDINGS OF THE TENTH SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES* (ACM SIGOPS), Orcas Island, Washington (December 1985). (Also released as technical report TR 85-668.)

[Birm87a] Birman, K. P. and T. A. Joseph. "Exploiting Virtual Synchrony in Distributed Systems." TECHNICAL REPORT TR 87-811, Department of Computer Science, Cornell University, Ithaca, NY, February 1987.

[Blac86b] Black, A., N. Hutchinson, E. Jul, and H. Levy. "Object Structure in the Emerald System." TECHNICAL REPORT 86-04-03, Department of Computer Science, University of Washington, Seattle, WA, April 1986.

[Blac87a] Black, A., N. Hutchinson, E. Jul, H. Levy, and L. Carter. "Distribution and Abstract Types in Emerald." *TRANSACTIONS ON SOFTWARE ENGINEERING* (IEEE)

13, no. 1 (January 1987). (Also available as University of Washington Technical Report 85-08-05.)

[Blac86a] Black, A. P., E. D. Lazowska, J. D. Noe, and J. Sanislo. "The Eden Project: A Final Report." TECHNICAL REPORT 86-11-01, Department of Computer Science, University of Washington, Seattle, WA, 1986.

[Coop84a] Cooper, E. "Circus: A Replicated Procedure Call Facility." *PROCEEDINGS OF THE FOURTH SYMPOSIUM ON RELIABILITY IN DISTRIBUTED SOFTWARE AND DATABASE SYSTEMS*, Silver Spring, MD (October 1984): 11-24.

[Coop85a] Cooper, E. "Replicated Distributed Programs." *PROCEEDINGS OF THE TENTH SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES* (ACM SIGOPS), Orcas Island, WA (December 1985): 63-78. (Available as *Operating Systems Review* **19**, no. 5.)

[Dasg88a] Dasgupta, P., R. J. LeBlanc, and W. F. Appelbe. "The Clouds Distributed Operating System: Functional Description, Implementation Details, and Related Work." *PROCEEDINGS OF THE EIGHTH INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS* (IEEE Computer Society), San Jose, CA (June 1988): 2-9. (Also available as Technical Report GIT-ICS-87/28.)

[El-A85a] El-Abbadi, A., D. Skeen, and F. Cristian. "An Efficient, Fault-Tolerant Protocol for Replicated Data Management." *PROCEEDINGS OF THE FOURTH SYMPOSIUM ON PRINCIPLES OF DATABASE SYSTEMS* (ACM SIGACT-SIGMOD) (March 1985).

[El-A87a] El-Abbadi, A. "A Paradigm for Concurrency Control Protocols." PH.D. DISS., Department of Computer Science, Cornell University, Ithaca, NY, 1987.

[Giff79a] Gifford, D. K. "Weighted Voting for Replicated Data." *PROCEEDINGS OF THE SEVENTH SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES* (ACM SIGOPS), Pacific Grove, CA (December 1979).

[Good83a] Goodman, N., D. Skeen, A. Chan, U. Dayal, R. Fox, and D. Ries. "A Recovery Algorithm for a Distributed Database System." *PROCEEDINGS OF THE SECOND SYMPOSIUM ON PRINCIPLES OF DATABASE SYSTEMS* (ACM SIGACT-SIGMOD), Atlanta, GA (March 1983).

[Grei86a] Greif, I., R. Seliger, and W. Weihl. "Atomic Data Abstractions in a Distributed Collaborative Editing System." *CONFERENCE RECORD OF THE THIRTEENTH SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES* (ACM SIGACT/SIGPLAN), St. Petersburg Beach, FL (January 1986). (Extended Abstract.)

[Herl84a] Herlihy, M. "Replication Methods for Abstract Data Types." PH.D. DISS., Laboratory for Computer Science, Massachussetts Institute of Technology, Cambridge, MA, May 1984. (Also released as Technical Report MIT/LCS/TR-319.)

[Herl85a] Herlihy, M. "Atomicity vs. Availability: Concurrency Control for Replicated Data." TECHNICAL REPORT CMU-CS-85-108, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, February 1985.

[Herl85b] Herlihy, M. "Using Type Information to Enhance the Availability of Partitioned Data." TECHNICAL REPORT CMU-CS-85-119, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, April 1985.

[Herl87a] Herlihy, M. P. and J. M. Wing. "Avalon: Language Support for Reliable Distributed Systems." *PROCEEDINGS OF THE SEVENTEENTH INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING*, Pittsburgh, PA (July 1987). (Also available as Technical Report CMU-CS-86-167.)

[Hone86a] Honeywell, Inc. "Fault Tolerant Distributed Systems." INTERIM SCIENTIFIC REPORT, Computer Sciences Center, Honeywell Inc., Golden Valley, MN, November 1986. (RADC Contract No. F30602-85-C-0300.)

[Jose85a] Joseph, T. A. "Low-Cost Management of Replicated Data." PH.D. DISS., Department of Computer Science, Cornell University, Ithaca, NY, November 1985. (Also released as Technical Report TR 85-712.)

[Jose86a] Joseph, T. A. and K. P. Birman. "Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems." *TRANSACTIONS ON COMPUTER SYSTEMS* (ACM) 4, no. 1 (Febuary 1986): 54-70.

[Kenl86a] Kenley, G. G. "An Action Management System for a Distributed Operating System." M.S. THESIS, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (Also released as technical report GIT-ICS-86/01.)

[LeBl85a] LeBlanc, R. J. and C. T. Wilkes. "Systems Programming with Objects and Actions." *PROCEEDINGS OF THE FIFTH INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS*, Denver (July 1985). (Also released, in expanded form, as technical report GIT-ICS-85/03.)

[LeLa78a] LeLann, G. "Algorithms for Distributed Data-Sharing Systems Which Use Tickets." *PROCEEDINGS OF THE THIRD BERKELEY WORKSHOP ON DISTRIBUTED DATA MANAGEMENT AND COMPUTER NETWORKS*, Berkeley, CA (August 1978).

[Lisk83a] Liskov, B. and R. Scheifler. "Guardians and Actions: Linguistic Support for Robust Distributed Programs." *TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS* (ACM) 5, no. 3 (July 1983).

[Lisk84a] Liskov, B. "Overview of the Argus Language and System." PROGRAMMING METHODOLOGY GROUP MEMO 40, Laboratory for Computer Science, Massachussetts Institute of Technology, Cambridge, MA, February 1984.

[Lisk86a] Liskov, B. and R. Ladin. "Highly-Available Distributed Services and Fault-Tolerant Distributed Garbage Collection." PROGRAMMING METHODOLOGY GROUP MEMO 48, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1986.

[Lisk87a] Liskov, B. and others. "Argus Reference Manual." PROGRAMMING METHODOLOGY GROUP MEMO NO. 54, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, March 1987.

[Lisk87b] Liskov, B. "Highly-Available Distributed Services." PROGRAMMING METHODOLOGY GROUP MEMO 52, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, February 1987.

[Long87a] Long, D. D. E. and J.-F. Paris. "On Improving the Availability of Replicated Files." *PROCEEDINGS OF THE SIXTH SYMPOSIUM ON RELIABILITY IN DISTRIBUTED SOFTWARE AND DATABASE SYSTEMS* (IEEE Computer Society), Williamsburg, VA (March 1987): 77-83.

[McKe85a] McKendry, M. S. "Ordering Actions for Visibility." *TRANSACTIONS ON SOFTWARE ENGINEERING* (IEEE) **11**, no. 6 (June 1985). (Also released as technical report GIT-ICS-84/05.)

[Noe85a] Noe, J. D., A. B. Proudfoot, and C. Pu. "Replication in Distributed Systems: The Eden Experience." TECHNICAL REPORT TR-85-08-06, Department of Computer Science, University of Washington, Seattle, WA, September 1985.

[Pitt86a] Pitts, D. V. "Storage Management for a Reliable Decentralized Operating System." PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (Also released as Technical Report GIT-ICS-86/21.)

[Pitt88a] Pitts, D. V. and P. Dasgupta. "Object Memory and Storage Management in the Clouds Kernel." *PROCEEDINGS OF THE EIGHTH INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS* (IEEE Computer Society), San Jose, CA (June 1988): 10-17.

[Prou85a] Proudfoot, A. B. "Replects: Data Replication in the Eden System." M.S. THESIS, Department of Computer Science, University of Washington, Seattle, WA, December 1985. (Also released as University of Washington Technical Report TR-85-12-04.)

[Schl83a] Schlichting, R. D. and F. B. Schneider. "An Approach to Designing Fault-Tolerant Systems." *TRANSACTIONS ON COMPUTER SYSTEMS* (ACM) **1**, no. 3 (August 1983): 222-238.

[Skee85a] Skeen, D. "Determining the Last Process to Fail." *TRANSACTION ON COMPUTER SYSTEMS* (ACM) 3, no. 1 (February 1985): 15-30.

[Spaf86a] Spafford, E. H. "Kernel Structures for a Distributed Operating System." PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (Also released as technical report GIT-ICS-86/16..)

[Ston79a] Stonebreaker, M. "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES." *TRANSACTIONS ON SOFTWARE ENGINEERING* (IEEE) **5**, no. 3 (May 1979).

[Stri88a] Strickland, H. "Networking Support for a Distributed Operating System." M.S. THESIS, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1988. (In progress.)

[Thom79a] Thomas, R. H. "A Majority Consensus Approach to Concurrency Control for Multiple-Copy Databases." *TRANSACTIONS ON DATABASE SYSTEMS* (ACM) **4**, no. 2 (June 1979).

[Weih83a] Weihl, W. and B. Liskov. "Specification and Implementation of Resilient Atomic Data Types." *SYMPOSIUM ON PROGRAMMING LANGUAGE ISSUES IN SOFTWARE SYSTEMS* (June 1983).

[Wilk85a] Wilkes, C. T. "Preliminary Aeolus Reference Manual." TECHNICAL REPORT GIT-ICS-85/07, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1985. (Last Revision: 17 March 1986.)

[Wilk86a] Wilkes, C. T. and R. J. LeBlanc. "Rationale for the Design of Aeolus: A Systems

Programming Language for an Action/Object System." *PROCEEDINGS OF THE 1986 INTERNATIONAL CONFERENCE ON COMPUTER LANGUAGES* (IEEE Computer Society), Miami, FL (October 1986): 107-122. (Also available as Technical Report GIT-ICS-86/12.)

[Wilk87a] Wilkes, C. T. "Programming Methodologies for Resilience and Availability." PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1987. (Also available as Technical Report GIT-ICS-87/32.)

[Wrig84a] Wright, D. D. "Managing Distributed Databases in Partitioned Networks." PH.D. DISS., Department of Computer Science, Cornell University, Ithaca, NY, January 1984. (Also available as Cornell University Technical Report 83-572.)

Appendix A

In this appendix, the Aeolus definition part for a resilient symbol table object is presented. This example is used in Section 4.

```
definition of recoverable object symtab
      ( name_type : type,  value_type : type ) is
   ! Single-copy symbol table object using the Aeolus/Clouds lock
   ! mechanisms for synchronization.  The definition part
   ! contains specifications of public constants, types, and
   ! operations defined by this object.  When compiled, it
   ! produces a symbol table file which may be imported by other
   ! objects using this object in their implementations.

operations

   procedure insert ( name  : name_type   ,
                      value : value_type  ,
                      error : out boolean ) modifies
      ! The insert operation places an entry into the
      ! symbol table.  error is set if an entry with the
      ! same name already exists.

   procedure delete ( name  : name_type   ,
                      error : out boolean ) modifies
      ! If the delete operation finds an entry with the
      ! given name, it removes the entry from the symbol table
      ! and frees its storage space.

   procedure lookup ( name  : name_type   ,
                      error : out boolean )
                    returns value_type examines
      ! The lookup operation tries to locate the entry with
      ! the given name and returns its value if it succeeds.
      ! error is set if the entry is not in the table.

   procedure quick_list () examines
      ! The quick_list operation provides a quick (dirty)
      ! listing of all names currently in the symbol table.

   procedure exact_list () examines
      ! The exact_list operation provides a listing of the
      ! exact state of the symbol table at a given point in time.
      ! To do this, it locks the whole symbol table, thereby
      ! excluding any changes during preparation of the listing.
      ! Thus, although exact_list, lookup, and
      ! quick_list operations may execute concurrently, and
      ! insert and delete operations which access
      ! different hash buckets may also execute concurrently,
      ! insert and delete operations must block on
      ! exact_list operations.

end definition.
```

Appendix B

In this appendix, the Aeolus definition part serving as the user interface to the Distributed Locking primitives is presented. This interface is discussed in Section 5.

```
definition of pseudo object DistLock is

    ! Interfaces to primitives provided for support of the
    ! Distributed Locking mechanism.  This pseudo-object is
    ! imported automatically by every availspec, and is not
    ! available for use by other compilands.

type replica_number is new unsigned
    ! A replica_number is used to name an individual replica of a
    ! group.  The naming scheme used here is the ''horizontal''
    ! method as described in Chapter VII of this dissertation.
    ! The replica_number is concatenated by the system to the
    ! capability of the object to which the invoking availspec
    ! belongs to form an extended capability as defined by the
    ! horizontal scheme.

type version_number is new longuns
    ! A version_number is used to compare the currency of
    ! the states of replicas.  The version number of an object is
    ! incremented whenever an invocation is performed on it, or
    ! when the state of the objected is updated by use of one of
    ! the designated operations described below.

operations

    procedure lock_replica ( rep : replica_number     ,
                             ver : out version_number )
            returns boolean modifies
        ! The lock_replica operation obtains the
        ! currently-requested lock at the replica denoted by rep.
        ! This operation should be invoked only within a lock event
        ! handler.  The lock variable, domain value, and mode
        ! requested are obtained from the context of the lock
        ! event which caused the invocation of the handler.
        ! The replica denoted by rep is added to a list of the
        ! replicas touched by the current action.
        ! The version number of the state of rep is returned
        ! in the out parameter ver.
        ! If lock_replica is unable to obtain the lock on
        ! rep, or if the requested lock is already held
        ! at rep by the current action, the operation returns
        ! FALSE, otherwise TRUE.
```

```
procedure invoke_replica ( rep : replica_number )
        returns boolean modifies
    ! The invoke_replica operation causes the current operation
    ! to be executed at the replica denoted by rep.  This
    ! operation should be invoked only within a copy event
    ! handler.  The operation number and other parameters are
    ! obtained from the context of the lock which caused the
    ! invocation of the handler.  The version number of rep
    ! is set to the value of that of the invoking object.
    ! This operation is used for implementing state copying by
    ! idemexecution.  If the invocation on rep is
    ! unsuccessful, the operation returns FALSE, otherwise
    ! TRUE.

procedure broadcast_shadows () returns boolean modifies
    ! The broadcast_shadow operation causes the ''shadow set''
    ! of the permanent state of the current action to be
    ! broadcast to all replicas at which locks were obtained by
    ! the current action via the lock_replica operation.
    ! The version numbers of the locked replicas are updated
    ! to equal that of the invoking object.  This
    ! operation should be invoked only within a copy event
    ! handler.  This operation is used for implementing state
    ! copying by cloning using shadows.  If all locked
    ! replicas successfully receive the shadow set, the
    ! operation returns TRUE, otherwise FALSE.

procedure get_state ( rep : replica_number )
        returns boolean modifies
    ! The get_state operation causes the state of the
    ! replica denoted by rep to be transmitted to the
    ! current object.  The state is installed at the current
    ! object, and its version number set to that of rep.
    ! If the transmission or installation fails, the operation
    ! returns FALSE, otherwise TRUE.

procedure send_state ( rep : replica_number )
        returns boolean modifies
    ! The send_state operation causes the state of the
    ! current object to be transmitted to the replica denoted
    ! by rep.  The state is installed at rep, and
    ! its version number set to that of the current object.  If
    ! the transmission or installation fails, the operation
    ! returns FALSE, otherwise TRUE.
    procedure invoke_acceptor ( rep   : replica_number ,
                                state : address        ,
                                len   : longuns        ) modifies
        ! The invoke_acceptor operation causes the invocation
        ! of the accept event handler at the replica denoted by
        ! rep.  The information the address of which is given by
        ! state and which is of length len bytes is copied to the
        ! environment of the accept handler at rep.  This operation
        ! may be used in a copy event handler to implement state
        ! copying by cloning using logs, or in a reinit event
        ! handler to implement state reconciliation strategies.

    procedure degree () returns replica_number examines
        ! The degree operation returns the total number of
        ! replicas of the current object including itself.

    procedure my_replica () returns replica_number examines
        ! The my_replica operation returns the replica number of
        ! the current object.

    procedure my_version () returns version_number examines
        ! The my_version operation returns the version number of
        ! the current object's state.

    procedure quorum_size () returns replica_number examines
        ! The quorum_size operation returns the minimum size of
        ! a quorum for the currently-requested lock mode.

    procedure currently_locked () returns replica_number
        ! The currently_locked operation returns the number of
        ! replicas on which the currently-requested lock mode has
        ! been obtained, including the current object.

end definition. ! DistLock
```

APPENDIX C

The Clouds Distributed Operating System: [†]
Functional Description, Implementation Details and Related Work.

Partha Dasgupta, Richard J. LeBlanc Jr., William F. Appelbe

Technical Report: GIT-ICS-87/42

Abstract

Clouds is an operating system in a novel class of distributed operating systems providing the integration, reliability and structure which makes a distributed computer system meet its hardware potential. *Clouds* is designed to run on a set of general purpose computers that are connected via a medium-to-high speed local area network. The structure of *Clouds* promotes transparency, support for advanced programming paradigms, and integration of resource management, as well as a fair degree of autonomy at each site.

The design criteria for *Clouds* include integration of resources through location transparency; support for various types of atomic operations, including conventional transactions; advanced support for achieving fault tolerance, provisions for dynamic reconfiguration and an object based system architecture. The implementation has been tailored to be simple, efficient and adaptable.

The system structuring paradigm chosen for the *Clouds* operating system, after substantial research, is an object/thread model, with facilities for atomic operations. All instances of services, programs and data in *Clouds* are are encapsulated in objects. The concept of persistent objects does away with the need for file systems, and replaces it with a more powerful concept, namely the object system. Concurrency control, atomicity and recovery are handled within objects.

In this paper, we provide a functional description of the system. We describe the preliminary implementation and show the modifications that are in progress for the next implementation. We present an overview of the current research results as well as directions for future efforts.

[†] This research was partially supported by NASA under contract number NAG-1-430, and by NSF under contract numbers DCS-8316590 and CCR-8619886.

The Clouds Distributed Operating System
Functional Description,
Implementation Details
and
Related Work.

Partha Dasgupta, Richard J. LeBlanc Jr., William F. Appelbe
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332

1. Introduction

Clouds is a distributed operating system under development. The goal of the *Clouds* project is to develop an instance of a class of distributed operating systems that provide the integration, reliability and structure that is necessary to make a distributed computing system effective.

Clouds is designed to run on a set of general purpose computers (uniprocessors or multiprocessors) that are connected via a medium-to-high speed local area network. The major design objectives for *Clouds* are:

- Integration of resources through cooperation and location transparency.
- Support for atomicity, transaction processing, and the ability to achieve fault tolerance (if needed).
- Efficient design and implementation.
- Simple and uniform interfaces for distributed processing.

The paradigm used for defining and implementing the software structure of the *Clouds* system, chosen after substantial research is an object/thread model. This model provides threads to support computation and objects to support an abstraction of storage. (These concepts are defined in sections 2 though 4). This model has been augmented to support atomicity of computation to provide support for reliable programs [A183, DaLe85, McA183]. In this paper, we provide a functional description of the system (sections 2 to 6), some implementational details (section 7), and discussion of related work (section 9).

1.1. Current Status

The first version of the *Clouds* operating system has been implemented and is operational. This version is referred to Clouds v.1. This is being used as an experimental testbed by the implementors.

Some of the performance figures for *Clouds* v.1. were:

Local Object Invocations	10 msec
Remote Object Invocations	40 msec

Commit of 1 page data	180 msec

The performance of our first verion is poor, due to several factors. The VAX architecture was not very suitable for implementing objects, and flushing of the translation buffers for each invocation causes the local invocation to be more expensive than expected. The Ethernet hardware used, coupled with a non-optimized driver gave us poor performance on round trip messages and hence large remote invocation times. The disk used in the commit tests was also exceedingly slow (40msec seek, 25msec/page write.) However, the experience with this version has taught us that the approach works. It also taught us how to do it better.

The lessons learned from this implementation are being used to redesign the kernel and build a new version. The basic system paradigm, the semantics of objects and threads and the goals of the project remain unchanged and v.2. will be identical to v.1. in this respect.

The structure of *Clouds* v.2. is different. The operating system will consist of a minimal kernel called "*Ra*". Ra will support the basic function of the system, that is location independent object invocation. The operating system will be built on top of the Ra kernel using system level objects to provide systems services (user object management, synchronization, naming, atomicity and so on.)

2. Objects

All data, programs, devices and resources on *Clouds* are encapsulated in entities called objects. The only entity recognized by the system, other than an object, is a thread. A *Clouds* object, at the lowest level of conception, is a virtual address space. Unlike virtual address spaces in conventional operating systems, a *Clouds* object is neither tied to any process nor volatile. A *Clouds* object exists forever (like a file) unless explicitly deleted. As will be obvious in the following description of objects, *Clouds* objects are somewhat 'heavyweight', that is they are suited for storage and execution of large-grained data and programs. This is because invocation and storage of object bear some non-trivial overhead.

Every *Clouds* object is named. The name of an object, also known as its *capability*, is unique over the entire distributed system and does not include the location of the object. That is, the capability-based naming scheme in *Clouds* creates a uniform, flat system name space for objects.

An object consists of a named address space and the contents of the address space. Since it does not contain a process, it is completely passive. Hence, unlike objects in some object based systems, a *Clouds* object is not associated with any server process. (The first system to use passive objects, though in a multiprocessor system was Hydra [Wu74, WuLe81]).

Threads are the active entities in the system, and are used to execute the code in an object (details in sections 2 and 3). A thread executes in an object by entering it through one of several entry points, and after the execution is complete the thread leaves the object. Several threads can simultaneously enter an object and execute concurrently.

Objects have structure. They contain, minimally, a code segment, a data segment and a mechanism for extending limits of storage allocated to the object. Threads that enter an object execute in the code segment. The data segment is accessible by the code in the code segment, but not by any other object. Thus the object has a wall around it which has some well-defined gateways, through which activity can come in. Data cannot be transmitted in or out of the object freely, but can be moved as parameters to the code segment entry points.

Clouds objects can be defined by the user or defined by the system. Most objects are user-defined. Some examples of system-defined objects are device drivers, name-service handlers, communication systems, systems software, utilities, and so on. The basic kernel (*Ra*) is not an object; it is an entity that provides the support for object invocation. A complete *Clouds* object can contain user-defined code and data; system-defined code and data that handle synchronization, recovery and commit; a volatile heap for temporary memory allocation; a permanent heap for allocating memory that will remain permanent as a part of the data structures in the object; locks; and capabilities to other objects.

Files in conventional systems can be conceived of a special case of a *Clouds* object. Thus, *Clouds* need not support a file system, but uses an object system. This is discussed in further detail in section 4.

Though *Clouds* object can be created, deleted and manipulated individually, the operating system is designed to support a class and instantiation mechanism. An object in the system can be an instance of its *template*. An object of a certain type is created by invoking a 'create' operation on the template of this type. Each template is created by invoking a create operation on a single template-template, which can create any template, if provided, as argument, the code and data definitions of the template. The templates, the template-template and all the instances thereof, are regular *Clouds* objects, and, as discussed earlier, they exist from the time of creation, until explicitly deleted.

3. Threads

The only form of activity in the *Clouds* system is the thread. A thread can be viewed as a thread of control that executes code in objects, traversing objects as it executes. Threads can span objects and machine boundaries. In fact, machine boundaries are invisible to the thread (and hence to the user). Threads are implemented in the *Clouds* system as lightweight processes, comprising of a PCB and a stack (but no virtual space). A thread that spans machine boundaries is implemented by several processes, one per site.

Upon creation, a thread starts up at an entry point of an object. As the thread executes, it executes code inside an object and manipulates the data inside this object. The code in the object can contain a procedure call to an operation of another object. When a thread executes this call, it temporarily leaves the caller object and enters the called object, and commences execution there. The thread returns to the caller object after the execution in the called object terminates. The calls to the entry point of objects are called *object invocations*. Object invocations can be nested. The code that is accessible by each entry point is known as an *operation* of the object.

A thread executes by processing operations defined inside many objects. Unlike processes in conventional operating systems, the thread often cross boundaries of virtual address spaces. Addressing in an address space is, however, limited to that address space, and thus the thread cannot access any data outside an address space. Control transfer between address spaces occurs though object invocation, and data transfer between address spaces occurs through parameters to object invocation.

When a thread executing in an object (or address space) executes a call to another object, it can provide the called operation with arguments. When the called operation terminates, it can send back result arguments. That is, object invocations may carry parameters in either direction.

These arguments are strictly data. Note that names (capabilities) are data. They may not be addresses. This restriction is necessary as the address space of each object may be disjoint, and an addresses is meaningful only in the context of the appropriate object. Parameter passing uses the copy-in-copy-out method.

4. The Object/Thread Paradigm

The structure created by a system composed of objects and threads has several interesting properties.

First, all inter-object interfaces are procedural. Object invocations are equivalent to procedure calls on modules not sharing global data. The modules are permanent. The procedure calls work across machine boundaries. (Since the objects exists in a global name space, there is no concept of machine boundaries.) Although local invocations and remote invocations (also known as remote procedure calls or RPCs) are differentiated by the operating system, this is transparent to the applications and systems programmers.

Second, the storage mechanism used in the object-based world is quite different from that used in the conventional operating systems. Conventionally, the file is the storage medium of choice for data that has to persist, especially since memory is tied to processes and processes can die and lose all the contents of their memory. However, memory is easier to manage, more suited for structuring data and essential for processing. The object concept merges these two views of storage, and creates the permanent virtual space.

For instance, a conventional file is a special case of an object. That is, a file is an object with operations such as read, write, seek, and so on, defined in it. These operations transport data in and out of the object through parameters provided to the calls.

Though files can be implemented using objects, the need for having files disappear in most situations. Programs do not need to store data in file-like entities, since they can keep the data in the data space in each object, structure appropriately. The need for user-level naming of files transforms to the need for user-level naming for objects.

Just as *Clouds* does not have files, it does not provide user-level support for file (or disk) I/O. In fact there is no concept of a "disk" or such I/O devices (except user terminals). The system creates the illusion of a huge virtual memory space that is permanent (non-volatile), and thus the need for using disk storage from a programmer's point of view, is eliminated.

Messages are a paradigm of choice in message-based distributed systems. In this case, like the need for I/O, the need for messages is eliminated. Threads need not communicate through messages. Thus ports are not supported. This allows a simplified system management strategy as the system docs not have to maintain linkage information between threads and ports.

Just as files can be simulated for those in need for them, messages and ports can be easily simulated by an object consisting of a bounded buffer that implements the send and receive operations on the buffer. However, we feel that the need for files and messages are the product of the programming paradigms designed for systems supporting these features, and these are not necessary structuring tools for programming environments.

A programmer's view of the computing environment created by *Clouds* is apparent. It is a simple world of named address spaces (or objects). These object live in computing systems on a LAN, but the machine boundaries are made transparent, creating a unified object space. Activity is provided by threads moving around amongst the population of objects through invocation; and data flow is implemented by parameter passing. The system thus looks like a set of permanent address spaces which support control flow through them, constituting what we term *object memory*.

This view of a distributed system does have some pitfalls. However these problems can be dealt with using simple techniques (implemented by the system), which are outlined below.

Threads aborting due to errors will leave permanent faulty data in objects they have modified. Failure of computers will result in similar mishaps. Multiple threads invoking the same object will cause errors due to race conditions and conflicts. More involved consistency violations may be the results of non-serializable executions. In a large distributed system, having thousands of objects and dozens of machines, corruption due to failure cannot be tolerated or easily repaired. The prevention of such situations is achieved through the use of atomicity at the processing level (not necessarily atomic actions). The following section gives a brief overview of the atomicity properties supported by *Clouds*.

5. Atomicity

The action support is an area where the *Clouds* v.1. and *Clouds* v.2. differ.

5.1. Actions in Clouds v.1.

In the first design, *Clouds* supported atomic actions and nested actions somewhat based on the model defined by Moss in his thesis [Mo81]. *Clouds* v.1. extended Moss's model by allowing custom tailored synchronization and recovery, as well as interactions between actions and non-actions.

The synchronization and recovery properties can be localized in objects, on a per object basis. The synchronization and recovery can be handled by the system (to adhere to Moss's semantics) or can be tailored by the user and thus provide facilities beyond those allowed by standard nested transactions. Customization is allowed by labeling of objects as "auto-sync" or "custom-sync" and "auto-recoverable" and "custom-recoverable".

When an object is "auto-sync" and "auto-recoverable" the system automatically provides 2-phase locking (at the object level) when threads executing on behalf of actions enter this object. When the action that touched the object commits, all objects are updated using the 2-phase commit protocol.

Custom synchronization and recovery can be used by applications for user programmed concurrency control and recovery. For this purpose the programmer has tools such as locks with arbitrary compatibilities, recoverable segments and per-action variables. Further details of this scheme can be found in [Wi87].

5.2. Atomicity in Clouds v.2.

The support for atomicity in *Clouds* v.2. has is roots in the above scheme, but has been changed in some respects. The following is a brief outline of the scheme. The actual methods used are discussed in greater detail in [ChDa87].

Instead of mandating customization of synchronization and recovery for application that cannot use strict atomicity semantics, the new scheme support a variety of *consistency preserving* mechanisms. The threads that execute are are of two kinds, namely s-thread (or standard threads) and cp-threads (or consistency preserving threads). The s-threads have a best effort execution scheme and are not provided with any system-level locking or recovery. The cp-threads on the other hand are supported by locking and recovery schemes, provided by the system. When a cp-thread executes, all pages it reads are read-locked and the pages it updates are write-locked. The updated pages are written using a 2-phase commit mechanism when the cp-thread completes.

The data in the system have an *instantaneous* version and a *stable* version. In fact, is nested threads are used, the data have a stack of versions, the top being the instantaneous version and the bottm being the stable version. All the threads work on the instantenous version. The data updated by cp-threads are committed when the cp-thread exits, while the data touched by the s-threads are committed "eventually", using a best effort semantics.

The cp-threads are allowed to interleave with s-threads, and also the cp-threads can be used to provide heavyweight as well as lightweight atomicity, using gcp and lcp operations, described below.

All threads are s-threads when created. The handling of cp-threads are programmed by the following scheme. All operations in objects in *Clouds* are tagged with a consistency label, the labels used are:

- Globally-Consistent (gcp)
- Locally-Consistent (lcp)
- Standard (s)
- Inherited (i)

An object can have any number of different labels on the operations. Also the same operation may have multiple entry points, labeled at different atomicity levels.

A s-thread executing a gcp or lcp operation converts to a cp-thread. A thread entering a lcp entry point, commits its updates (inside this object) as soon as it exits the object. This provides intra-object consistency rather than the inter-object consistency provided by the gcp operations, and thus is a cheap method of updating one object atomically. Locking and recovery are automatic.

The standard entry points do not support any locking or recovery. They can make use of "best-effort" semantics. They can also be used for non-traditional purposes such as peeking at incomplete results of actions (as they are not hindered by locking and visibility rules of actions). Locks are available for synchronizing non-actions, but recovery is not supported.

The other labels as well as combination of these labels in the same object (or in the same thread) lead to many interesting (as well as dangerous) variations. The complete discussion of the semantics as well as the implementation is beyond the scope of this paper, and the reader is referred to [ChDa87]

6. Programming Support

Systems and application programming for *Clouds* involves programming objects that implement the desired functionality. These objects can be expressed in any programming language. The compiler (or the linker) for the language, however, must be modified to generate the stubs for the various entry points, invocation handler, system call interfaces and the inclusion of default systems function handling code (such as synchronization and recovery). Most of *Clouds* application programming have been done in a pre-processed form of C, that we call C-weed.

The language Aeolus has been designed to integrate the full set of powerful features that the *Clouds* kernel supports. Aeolus currently supports the features of *Clouds* v.1. but is being expanded for added functionality of Ra and *Clouds* v.2. [LeWi85, Wi85, WiLe86].

Aeolus is the first generation language for *Clouds*. It does not support some of the features found in object-oriented programming systems such as inheritance and subclassing. Providing support for these features at the language level is currently under consideration.

7. Enhancements and Planned Features

The above description of *Clouds* documents the basic features of the distributed kernel for *Clouds*. Presently the following enhancement, applications and features are at various stages of design, implementation and planning.

- An object naming scheme is being developed that creates a hierarchical user naming strategy (like Unix) that is also highly available and robust (through replicated directories).
- Unix and *Clouds* will be inter-operable providing Unix programmers and user with access to *Clouds* features and *Clouds* programmers to use Unix services. Unix machines will be able to execute remote procedure calls to *Clouds* object thus gaining access to all the functionality that *Clouds* provides. In fact the user interface to *Clouds* can be achieved (initially) through Unix shells and tools. Similarly *Clouds* applications can make use of the wide variety of programming support tools that are supported by Unix through a mechanisms that provides Unix service for *Clouds* computations.
- As mentioned earlier, mechanisms for providing object-oriented programming methodology will be provided at the linguistic level, with enhancements in the kernel that will provide performance advantages (such as sharing of code in the classes with its instances).
- Debugging support at the object level, thread level and the invocation level is necessary. Techniques that allow the programmer to get a comprehensive view of the distributed and concurrent execution environment are under development.

- A probe system that can track object and thread status in the system can provide information about failures, loading, deadlocks and software problems is being developed. This will be used to develop a distributed system monitoring system that will help in reconfiguration on failure and aid in providing fault tolerance. The probe system will also be useful in distributed object level debugging [Da86].
- A distributed database that utilizes the object structure of *Clouds* for elegance and the synchronization and recovery support for concurrency control and reliability is being developed [DaMo86].
- *Clouds* has been designed as an operating system, that can support fault tolerance computing. The systems that will provide fault tolerance and guarantee progress of computation and system response in face of partial system failures are being developed. The techniques include replicated objects, multi-threaded actions, the coupling of the reconfiguration systems and monitoring systems.

8. Implementation Notes

The implementation of the *Clouds* operating systems has been based on the following guidelines:

- The implementation of the system should be suitable for general purpose computers, connected through common networking hardware. Heterogeneous machines, though not crucial, should be allowed.
- Since the *Clouds* functionality is largely based on object invocation, support for objects should be efficient in order to make the system usable. Also, the naming, synchronization and recovery systems should be implementable with minimal overhead.
- Since one of the primary aims of *Clouds* is to provide the substrate for reliable, fault tolerant computing, the kernel and the operating system should provide adequate support for implementing fault tolerance.
- The system design should be simple to comprehend and implement.

8.1. Hardware Configuration

Clouds v.1. was built on a three VAX-11/750 computers, connected through an Ethernet, equipped with RL02 and RA81 disk drives. The user interface was through the Ethernet, accessible from any Unix machine.

Clouds v.2. is being implemented on a set of Sun-3 class machines. The cluster of Clouds machines are on an Ethernet, and users will access them through workstations running *Clouds* as well as Unix workstations.

8.2. Software Configuration and Kernel Structure

The kernel (version 2.) used to support *Clouds* is called *Ra*. *Ra* is a native kernel running on bare hardware. The kernel is implemented in C for portability, and because the availability of C source for UNIX kernels simplified the task of developing hardware interfaces such as device drivers.

The kernel runs on the native machine and not on top of any conventional operating system for two reasons. Firstly, this approach is efficient. As *Clouds* does not use much of the functionality of conventional operating systems (such as file systems), building *Clouds* on top of a Unix-like kernel make poor use of the host operating system. Secondly, the paradigms and the support for synchronization, recovery, shared memory and so on; used in *Clouds* are considerably different from the functionality provided by conventional operating systems, and major changes would be necessary at the kernel level of any operating system in order to implement *Clouds*.

The Ra kernel provides support for partitions, segments, virtual spaces, processes and threads. These are the basic building blocks for Clouds. The partitions provide non-volatile storage, the segments provide memory storage, which are used to build objects, which in turn reside in virtual spaces. Processes provide activity which are used to compose threads. A description of the design of Ra can be found in [BeHuKh87]

8.3. Object Naming and Invocation

The two basic activities inside the Ra kernel are system call handling and object invocations. System call handling is done locally, as in any operating system. The system calls supported by the Ra kernel include object invocation, memory allocation, process control and synchronization, and other localized systems functions. Object invocation is a service provided by the kernel for user threads. The attributes that object invocation satisfy are:

- Location independence.
- Fast, for both local and remote invocations.
- Failed machines should not hamper availability of objects on working sites, from working sites.
- Moving objects between sites, reassigning disk units and so on should be simple (for fault tolerance support).

Location independence is achieved through a capability based naming system. Availability is obtained through decentralization of directory information and a search-and-invoke strategy coupled with a multicast based object location scheme, designed for efficiency [AhAm87]. Speed is achieved by implementing the invocation handlers at the lowest level of the kernel, on the native machine.

8.4. Storage Mangement

The storage management system handles the function required to provide the reliable, permanent object address spaces. As mentioned earlier, unlike conventional systems, where virtual address spaces are volatile and short-lived, *Clouds* virtual spaces contain objects and are permanent and long lived. The first version of the implementation is detailed in [Pi86].

The storage management system stores the object representations on disk, as an image of the object space. When an object is invoked, the object is demand paged into its virtual space as and when necessary. As the invocation updates the object, the updated pages do not replace the original copy, but have shadow copies on the disk. The permanent copy is updated only when a commit operation is performed on the object. The storage manager provides the support to commit an object using the two-phase commit protocol.

9. Comparisons with Related Systems

Clouds is one of the several research projects that are building object-based distributed environments. Although there are differences between all the approaches, we feel that the area of distributed operating systems is not mature enough to conclusively argue the superiority of one approach over the other. In the following paragraphs we document the major differences between *Clouds* and some of the better known projects in distributed systems. (This list is not exhaustive).

One of the major difference between *Clouds* and some of the systems mentioned below is in the implementation of the kernel. Many systems implement the kernel as a Unix process[†], while *Clouds* is implemented as a native operating system (as are Mach and Alpha). *Clouds* is not intended to be an enhancement, or replacement of, the UNIX kernel. Instead, *Clouds* provides a different paradigm from that supported by UNIX (e.g., the UNIX paradigms of 'devices as files', unstructured files, volatile address spaces, pipes, redirection etc.)

9.1. Argus

Argus is a language for describing objects, actions and processes using the concept of a guardian. The language defines a distributed system to be a set of guardians, each containing a set of handlers. Guardians are logical sites, and each guardian is located at one site, though a site may contain several guardians. The handlers are operations that can access data stored in the guardian. The data types in Argus can be defined to be atomic, and atomic data types changed by actions are updated atomically when the action terminates [WeLi83, LiSc83]. The support for Argus is built on top of Unix, and provides all the facilities of the Argus language [Li87].

Some of the similarities between Argus and *Clouds* are in the semantics of nested actions. Both use the nested action semantics and locking semantics that are derived from Moss. This includes conditional commit and lock inheritance. However the consistency preserving mechanisms in *Clouds* have moved away from Moss's action semantics, substantially, though retaining the nested action semantics as a subset. Also the guardians and handlers in Argus have somewhat more than cosmetic similarities to objects in *Clouds*, as the design of *Clouds* was influenced by Argus.

The differences include the implementation strategies, programming support and support for reliability. The scheme of permanent virtual spaces provided by passive objects is a major difference. As mentioned earlier, Argus is implemented on top of a modified Unix environment. This is one of the reasons for the somewhat marginal performance of the Argus system observed in [GrSeWe86]. The programming support provided by Argus is for the Argus language. *Clouds* on the other hand is a general purpose operating system, not tied to any language. Though Aeolus is the preferred language at present, we have used C extensively for object programming. We have plans to implement more object-oriented languages for the the *Clouds* system.

[†]The term ***kernel*** has been used quite frequently to describe the core service center of a system. However when this service is provided by a Unix process rather than a resident, interrupt driven monitor, the usage of the term is somewhat counter-intuitive.

9.2. Eden

Eden is a object-based distributed system, implemented on the Unix operating system at the University of Washington. Eden objects (called Ejects) use the active object paradigm, that is each object consists of a process and an address space. An invocation of the object consists of sending a message to the (server) process in the object, which executes the requested routine, and returns the results in a reply [Alm83, AlBl83, NoPr85].

Since every object in the system needs to have a process servicing it, this could lead to too many processes. Thus Eden has an *active* and a *passive* representation of objects. The passive representation is the core image of the object stored on the disk. When an object is invoked, it must be active, thus invoking a passive object involves activating it. A process is created by 'exec'-ing the core image of the object (frozen earlier), and then performs the required operation. The activation of passive objects is an expensive operation. Also concurrent invocations of objects are difficult and are handled through multithreaded processes or coroutines.

The active object paradigm and the Unix-based implementation are some of the major differences between Eden and *Clouds*. Eden also provides support for transaction and replication objects (called Replects). The transaction support and replication were added after the basic Eden system was designed and have some limitations due to manner Unix handles disk I/O.

9.3. Cronus

Cronus is an operating system designed and implemented at BBN Laboratories. Some of the salient points of Cronus are the intergration of Cronus functions with Unix functions, the ability of Cronus to handle a wide variety of hardware and the coexistence of Cronus on a distributed set of machines running Unix, as well as several other host operating systems [BeRe85, GuDe86, ScTh86].

Like Eden, Cronus uses the active objects. This is necessary to be able to make Cronus run on top of most host operating systems. Cronus objects are handled by managers. Often a single manager can handle several objects, by mapping the objects into its address space. The managers are servers and receive invocation requests through catalogued ports. Any Unix process on any machine on the network can avail of Cronus services from any manager, by sending a message to the appropriate manager. By use of canonical data forms, the machine dependencies of data representations are made transparent. Irrespective of the machine types, any Unix machine can invoke Cronus objects in a location independent fashion.

9.4. ISIS

ISIS (version 1) is a distributed operating system, developed at Cornell University, to support fault tolerant computing. ISIS has been implemented on top of Unix. It uses replication and checkpointing to achieve failure resilience. If data object is declared to be k-resilient, the system creates k+1 copies of the object. The replicated object invocation is handled by invoking one replica and transmitting the state updates to all replicas. Checkpointing at each invocation is used to recover from failures [Bi85A, Bi85B].

The goals and attributes of ISIS are different from Clouds. ISIS is built on top of some interesting communication primitives and is not built as a general purpose computing environment.

9.5. ArchOS and Alpha

Alpha is the kernel for the ArchOS operating system developed by the Archons project at Carnegie Mellon University. Like *Clouds*, the Alpha kernel is a native operating system kernel designed to run on the special hardware called Alpha-nodes. The Alpha kernel uses passive objects residing in their own virtual spaces, similar to *Clouds*. ArchOS is designed for real time applications supporting specialized defense related systems and applications [Je85, No87].

The key design criteria for ArchOS and Alpha are time critical computations and rather than reliability. Fault tolerance is handled to an extent using communication protocols. Real time scheduling has been a major research topic at the Archons project.

9.6. V-System

The V operating system has been developed at Stanford University. V is a compromise between message-based systems and object-based systems. The basic core of V provides lightweight processes and a fast communications (message) system. V message semantics are similar to object invocations in the sense that the messages are synchronous and use the send/reply paradigm. The relationship between processes conforms to the client-server paradigm. A client sends a request to the server, and the client blocks until the server replies [ChZw83].

V allows multiple processes to reside in the same address space. Data sharing is through message passing, though shared memory can be implemented through servers managing bounded buffers. The design goals of V are primarily speed and simplicity. V does not provide transaction and replication support. These can be implemented, if necessary at the application level.

The radical difference between V and *Clouds* is the paradigm used by *Clouds*.

9.7. Mach

Mach is a distributed operating system under development at Carnegie Mellon. Mach maintains object-code compatibility with Unix. Mach extends the Unix paradigms by adding large sparse address spaces, memory mapped files, user provided backing stores, and memory sharing between tasks. Mach is implemented on a host of processors including multiprocessors.

The execution environment for a Mach activity is a task. Threads are computation units that run in a task. A single thread in a task is similar to a Unix process. Ports are communication channels, supporting messages which are typed collection of data objects. In addition, Mach supports memory objects, which are collections of data objects managed by a server.

Support for transactions are not built into Mach, but can be layered on top of Mach and has been implemented by Camelot and Avalon [HeWi87].

The approaches used by Mach and *Clouds* are fundamentally different, as with V and *Clouds*.

10. Concluding Remarks

Clouds provides an environment for research in distributed applications. By focusing on support for advanced programming paradigms, and decentralized, yet integrated, control, *Clouds* offers more than 'yet another Unix extension/look-alike'. By providing mechanisms, rather than policies, for advanced programming paradigms, *Clouds* provides systems researchers a adaptable, high-performance, 'workbench' for experimentation in areas such as distributed databases, distributed computation, and network applications. By adopting 'off the shelf' hardware, the portability and robustness of *Clouds* are enhanced. By providing a 'Unix gateway', users can make use of established tools. The gateway also relieves *Clouds* from the necessity of providing emulating services such as provided by Unix mail and text processing.

The goal of *Clouds* has been to build a general purpose distributed computing environment, suitable for a wide variety of user communities, both within and outside the computer science community. We are striving to achieve this through a simple model of a distributed environment with facilities that most users would feel comfortable with. Also we are planning to experiment with increased usage of the system by making it available to graduate courses, and hope the feedback and the criticism we receive from a large set of users will allow us to tailor, enhance and perhaps redesign the system to fit the needs for distributed computing, and thus give rise to wider usage of distributed systems.

11. Acknowledgements

The authors would like to acknowledge Martin McKendry and Jim Allchin for starting the project and designing the first version of Clouds. Gene Spafford and Dave Pitts for the implementation, Jose Bernabeau, Yousef Khalidi and Phil Hutto for their efforts in making the kernel usable and for the design of *Ra*. Also Mustaq Ahamad, Ray Chen, Kishore Ramachandran and Henry Strickland for their participation in the project.

12. References

[Ac86] Accetta M, et. al. *Mach: A New Kernel Foundation for Unix Development*, Technical Report, Carnegie Mellon University.

[AhAm87] M. Ahamad, M. Ammar, J. Bernabeu and M. Y. Khaldi, *A Multicast Scheme for Locating Objects in a Distributed System*. Technical Report GIT-ICS-87/01, School of Information and Computer Science, Georgia Tech, January 1987.

[Alm83] G. T. Almes, *The Evolution of the Eden Invocation Mechanism*, Technical Report 83-01-03, Department of Computer Science, University of Washington, 1983.

[Al83] J. E. Allchin, *An Architecture for Reliable Decentralized Systems*, Ph.D. Diss., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, (Also released as technical report GIT-ICS-83/23,) 1983.

[AlBl83] G. T. Almes, A. P. Black and E. D. Lazowska and J. D. Noe, *The Eden System: A Technical Review*, University of Washington Department of Computer Science, Technical Report 83- 10-05 October 1983.

[AlMc82] J. E. Allchin and M. S. McKendry, *Object-Based Synchronization and Recovery*, Technical Report GIT-ICS-82/15 School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1982.

[BeHaKh87]
J. M. Bernabeau Auban, P. W. Hutto and M. Y. A. Khalidi, *The Architecture of the Ra Kernel*, Technical Report GIT-ICS-87/35 School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1987.

[BeRe85] J. C. Berets, R. A. Mucci and R. E. Schantz, *Cronus: A Testbed for Developing Distributed Systems*, October 1985 IEEE Communications Society, IEEE Military Communications Conference.

[Bi85A] K. P. Birman and others, *An Overview of the ISIS Project*, Distributed Processing Technical Committee Newsletter, IEEE Computer Society (7,2) October 1985 (Special issue on Reliable Distributed Systems).

[Bi85B] K. P. Birman, *Replication and Fault-Tolerance in the ISIS System*, ACM SIGOPS, Proceedings of the Tenth Symposium on Operating Systems Principles, December 1985 Orcas Island, Washington, (Also released as technical report TR 85-668).

[ChDa87] R. Chen and P. Dasgupta, *Consistency-Preserving Threads: Yeat Another Approach to Atomic Programming*, Technical Report GIT-ICS-87/43 School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1987.

[ChZw83] D. R. Cheriton and W. Zwaenepoel, *The Distributed V Kernel and its Performance for Diskless Workstations*, Proceedings of the Ninth Symposium on Operating Systems Principles, ACM SIGOPS, Bretton Woods, NH, October 1983.

[Da86] P. Dasgupta, *A Probe-Based Fault Tolerant Scheme for an Object-Based Operating System*, Proceeding of the 1st ACM Conference on Object Oriented Programming Systems, Languages and Applications. Portland OR. 1986.

[DaLe85] P. Dasgupta, R. LeBlanc and E. Spafford, *The Clouds Project: Design and Implementation of a Fault-Tolerant Distributed Operating System*, Technical Report GIT-ICS-85/29, 1985 School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA.

[DaMo86] P. Dasgupta and M. Morsi, *An Object-Based Distributed Database System Supported on the Clouds Operating System*, Technical Report GIT-ICS-86/07, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986.

[GuDe86] R. F. Gurwitz, M. A. Dean and R. E. Schantz, *Programming Support in the Cronus Distributed Operating System*, May 1986, Proceedings of the Sixth International Conference on Distributed Computer Systems, IEEE Computer Society.

[GrSeWe86]
I. Greif, R. Seliger and W. Weihl *Atomic Data Abstractions in a Distributed Collaborative Editing System*, (Extended Abstract) Conference Record of the Thirteenth Symposium on Principles of Programming Languages, ACM SIGACT/SIGPLAN, January 1986, St. Petersburg Beach, FL.

[HeWi87] M. P. Herlihy and J. M. Wing, *Avolon: Language Support for Reliable Distributed Systems*. Proceedings of the 17th International Symposium on Fault-Tolerant Computing. July 1987.

[Je85] E. D. Jensen et. al. *Decentralized System Control*, Technical Report RADC-TR-85-199, Carnegie Mellon University and Rome Air Development Center, April 1985.

[Ke86] G. G. Kenley, *An Action Management System for a Distributed Operating System*, M.S. Thesis, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (Also released as technical report GIT-ICS-86/01).

[LeWi85] R. J. LeBlanc and C. T. Wilkes, *Systems Programming with Objects and Actions*, Proceedings of the Fifth International Conference on Distributed Computing Systems, Denver, July 1985. (Also released, in expanded form, as technical report GIT-ICS-85/03)

[LiSc83] B. Liskov and R. Scheifler, *Guardians and Actions: Linguistic Support for Robust Distributed Programs*, ACM, Transactions on Programming Languages and Systems (53) July 1983.

[Li87] B. Liskov, D. Curtis, P. Johnson and R. Scheifer. *Implementation of Argus*. Proceedings of the 11th ACM Symposium on Operating Systems Principles. November 1987.

[Mc84A] M. S. McKendry, *Clouds: A Fault-Tolerant Distributed Operating System*, Distributed Processing Technical Committee Newsletter, IEEE, 1984, (Also issued as Clouds Technical Memo No:42).

[Mc84B] M. S. McKendry, *Fault-Tolerant Scheduling Mechanisms*, (Unpublished Technical Report), School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, May 1984, (Draft only).

[Mc85] M. S. McKendry, *Ordering Actions for Visibility*, Transactions on Software Engineering, IEEE (11,6) June 1985 (Also released as technical report GIT-ICS-84/05).

[McAl83] M. S. McKendry, J. E. Allchin and W. C. Thibault, *Architecture for a Global Operating System*, IEEE Infocom, April 1983.

[Mo81] J. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*, Technical Report MIT/LCS/TR-260, MIT Laboratory for Computer Science, 1981.

[MuMo83] E. T. Mueller, J. D. Moore and G. J. Popek, *A Nested Transaction Mechanism for LOCUS*, Proceedings of the Ninth Symposium on Operating Systems Principles, ACM SIGOPS, Bretton Woods, NH, October 1983.

[NoPr85] J. D. Noe, A. B. Proudfoot and C. Pu, *Replication in Distributed Systems: The Eden Experience*, Department of Computer Science, University of Washington, Seattle, WA, September 1985 Technical Report TR-85-08-06.

[No87] Northcutt J. D. *Mechanisms for Reliable Distributed Real-Time Operating Systems - The Alpha Kernel*, Perspectives in Computing, v16. Academic Press, 1987.

[Pi86] D. V. Pitts, *Storage Management for a Reliable Decentralized Operating System*, Ph.D. Diss., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986, (Also released as Technical Report GIT-ICS-86/21).

[ScTh86] R. E. Schantz, R. H. Thomas and G. Bono, *The Architecture of the Cronus Distributed Operating System*, May 1986, Proceedings of the Sixth International Conference on Distributed Computer Systems, IEEE Computer Society.

[Sp86] E. H. Spafford, *Kernel Structures for a Distributed Operating System*, Ph.D. Diss., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986, (Also released as technical report GIT-ICS-86/16).

[SpBu84] A. Z. Spector, J. Butcher, D. S. Daniels and others, *Support for Distributed Transactions in the TABS Prototype*, July 1984, Technical Report CMU-CS-84-132, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA.

[WaPo83] B. Walker, G. Popek, R. English, C. Kline and G. Thiel, *The LOCUS Distributed Operating System*, Proceedings of the Ninth Symposium on Operating Systems Principles, Bretton Woods, NH, ACM SIGOPS, pp. 49-70, October 1983. (Available as *Operating Systems Review* 17, no. 5)

[WeLi83] W. Weihl and B. Liskov, *Specification and Implementation of Resilient Atomic Data Types*, Symposium on Programming Language Issues in Software Systems, June 1983.

[Wi85] C. T. Wilkes, *Preliminary Aeolus Reference Manual*, Technical Report GIT-ICS-85/07, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1985. (Last Revision: 17 March 1986)

[WiLe86] C. T. Wilkes and R. J. LeBlanc, *Rationale for the Design of Aeolus: A Systems Programming Language for an Action/Object System*, Proceedings of the IEEE Computer Society 1986 International Conference on Computer Languages. (Also available as Technical Report GIT-ICS-86/012, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986.)

[Wi87] C.T. Wilkes *Programming Methodologies for Resilience and Availability*. Ph.D.thesis, Georgia Tech, 1987, Technical Report GIT-ICS-87/32 School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA.

[Wu74] W. A. Wulf and others, *HYDRA: The Kernel of a Multiprocessor Operating System*, Communications of the ACM, (17,6) June 1974.

[WuLe81] W. A. Wulf, R. Levin and S. P. Harbison, *HYDRA/C.mmp, An Experimental Computer System*, McGraw-Hill, Inc., 1981.